普通高等教育“十一五”国家级规划教材

DSP 芯片的原理与开发应用
（第 4 版）

张雄伟 曹铁勇 陈亮 杨吉斌 吴其前 等编著

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

内 容 简 介

本书由浅入深、全面系统地介绍了 DSP 芯片的基本原理、开发和应用。首先介绍了 DSP 芯片的基本结构和特征，以及定点和浮点 DSP 处理的运算基础；其次介绍了 DSP 芯片的开发工具，重点介绍了目前广泛应用的 CCS 集成开发环境及其使用方法；接着，介绍了基于 C 语言和汇编语言的开发方法以及 DSP 芯片的存储资源管理，较为详细地介绍了 DSP 系统的软硬件设计方法；最后，通过三个应用实例介绍了定点和浮点 DSP 芯片的系统开发过程，并介绍了数字滤波器和 FFT 等常用数字信号处理算法的 DSP 实现。

本书的目的是使读者了解 DSP 芯片的基本原理和常用 DSP 芯片的应用，熟悉 DSP 芯片开发工具及使用方法，掌握 DSP 系统的软硬件设计和应用系统开发方法，具备从事 DSP 芯片软硬件设计和系统开发的能力。

本书的特点是：结构清晰、内容全面、举例丰富、实用性强，可作为电子信息类学科的研究生和相关专业高年级大学生的教材，也可作为 DSP 芯片应用人员的培训教材，对于从事 DSP 芯片开发应用的科技人员和高校教师也具较高的参考价值。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

DSP 芯片的原理与开发应用 / 张雄伟，曹铁勇，陈亮等编著. —4 版. —北京：电子工业出版社，2009.3
（普通高等教育“十一五”国家级规划教材）
ISBN 978-7-121-07381-6

I. D… Ⅱ.①张…②曹…③陈… Ⅲ.数字信号－信号处理－高等学校－教材 Ⅳ.TN911.72

中国版本图书馆 CIP 数据核字（2008）第 140268 号

责任编辑：高买花　　特约编辑：陈宁辉
印　　刷：
装　　订：
出版发行：电子工业出版社　http://www.phei.com.cn
　　北京市万寿路 173 信箱　　邮编　100036
开　　本：787×1092　1/16　印张：26.5　字数：678 千字
版　　次：2009 年 3 月第 4 版　2009 年 3 月第 1 次印刷
印　　数：4 000 册　　定价：49.80 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

第 4 版说明

本书第 1 版（1997 年）、第 2 版（2000 年）和第 3 版（2003 年）深得广大读者的厚爱，受到了广泛的欢迎，广大 DSP 开发人员将本书作为 DSP 应用开发的重要参考书，很多开设 DSP 芯片原理与应用的高校将本书作为教材。2001 年，本书第 2 版获得全国优秀畅销书奖。2006 年，本书被列入普通高等教育“十一五”国家级规划教材。2007 年，本书作者被评为电子工业出版社优秀作者。对于广大读者对本书的热情支持，作者表示深深的谢意。

自本书第 2 版、第 3 版出版发行以来，又有许多高性能的 DSP 芯片和开发工具出现，很多高校也相继开设 DSP 芯片应用的课程。因此，在重新出版本书时，作者根据 DSP 芯片的发展，特别是为了便于利用本书实施课程教学，对本书的结构和内容作了较大程度的修订和补充，调整修改了有关章节的内容，补充完善了附录内容。

希望读者在使用本书的过程中提出宝贵的建议和意见，以便在今后的修订过程中参考。

本书得到解放军理工大学精品教材立项资助

本书得到江苏省“333 高层次人才培养工程”资助

前　言

DSP 芯片，也称数字信号处理器，是一种具有特殊结构的微处理器。DSP 芯片的内部采用程序和数据分开的哈佛结构，具有专门的硬件乘法器，广泛采用流水线操作，提供特殊的 DSP 指令，可以用来快速实现各种数字信号处理算法。

自 20 世纪 80 年代初 DSP 芯片诞生以来，DSP 芯片在近 30 年时间里得到了飞速的发展，DSP 芯片的性能价格比不断提高，开发手段越来越完善，应用越来越普及。DSP 芯片已经在通信与电子系统、信号与信息处理、自动控制、雷达、军事、航空航天、医疗、家用电器等许多领域得到广泛的应用。

DSP 芯片可分为通用型和专用型两大类。通用型 DSP 芯片是一种软件可编程的 DSP 芯片，适用于各种 DSP 应用场合。专用型 DSP 芯片则将 DSP 处理的算法集成到 DSP 芯片内部，一般适用于某些专用的场合。本书主要讨论通用型的可编程 DSP 芯片。

目前，DSP 芯片的主要供应商包括美国的德州仪器公司（TI）、AD 公司、AT&T 公司和 Motorola 公司等。其中，TI 公司的 DSP 芯片占世界 DSP 芯片市场近 50%，在国内也被广泛地采用。因此，本书主要以 TI 公司的 DSP 芯片为例进行介绍。

本书共 14 章，可分为 5 个部分。

第一部分是 DSP 芯片基础，包括第 1、2、3 章。第 1 章概述 DSP 芯片的发展、分类、选择和应用；第 2 章介绍 DSP 芯片的基本结构和 TI 等公司的 DSP 芯片的主要特征；第 3 章介绍 DSP 的数值运算基础，包括定点 DSP 处理中的定标、定点和浮点 DSP 处理的有关问题。

第二部分是软件开发基础，包括第 4、5 章。第 4 章比较详细地介绍 TI 公司推出的集成开发环境——CCS 的基本原理和使用方法；第 5 章介绍用 C 语言开发 DSP 芯片的方法。

第三部分是 DSP 芯片的资源管理和汇编语言开发，包括第 6、7、8 章。第 6 章介绍 DSP 芯片的存储资源管理，重点介绍广泛采用的公共目标文件格式（COFF）和编程方法；第 7 章重点介绍 TMS320C5000 系列芯片的汇编语言指令和寻址方式；第 8 章介绍基于 C 语言和汇编语言混合编程的 DSP 软件开发。

第四部分是 DSP 系统的硬件设计，包括第 9、10、11 章。第 9 章介绍 DSP 芯片的片内集成外设的开发方法；第 10 章介绍 DSP 脱机系统的设计；第 11 章介绍 DSP 系统的硬件设计方法。

第五部分是系统设计与应用实例，包括第 12、13、14 章。第 12 章以 TMS320C203、TMS320C31、TMS320VC5409 型芯片为例介绍 3 个应用系统的设计方法及开发过程；第 13 章介绍数字滤波器的 DSP 实现，包括 FIR、IIR 和自适应滤波器的定点和浮点 DSP 实现方法；

第 14 章介绍 FFT 的定点和浮点 DSP 实现。

本书由张雄伟主编，并对全书进行统稿。第 1、3 章由张雄伟编写；第 2、8 章由曹铁勇、张雄伟编写；第 4 章由吴其前、徐光辉编写；第 5、9 章由杨吉斌编写；第 6 章由张雄伟、邹霞编写；第 7 章由陈亮编写；第 10 章由曹铁勇、陈亮编写；第 11 章由杨吉斌、张雄伟编写；第 12 章由张雄伟、贾冲编写；第 13、14 章和附录由张雄伟、陈亮、李莉编写。在本书的编写过程中，许晔峰、徐志军、王金明、黄忠虎等提出了十分有益的建议，在此向他们表示衷心的感谢。

由于水平所限，书中错误之处在所难免，恳请广大读者给予批评指正。

编 著 者

2008 年 10 月于解放军理工大学

目　录

第 1 章　概述 ……（1）
1.1　引言 ……（1）
1.2　DSP 系统 ……（2）
1.2.1　DSP 系统的基本构成……（2）
1.2.2　DSP 系统的特点 ……（2）
1.2.3　DSP 系统的设计 ……（3）
1.2.4　DSP 系统的开发 ……（4）
1.3　DSP 芯片 ……（5）
1.3.1　DSP 芯片的定义 ……（5）
1.3.2　DSP 芯片的特点 ……（5）
1.3.3　DSP 芯片的发展 ……（6）
1.3.4　TI 公司的 DSP 芯片……（7）
1.3.5　DSP 芯片的分类 ……（8）
1.3.6　DSP 芯片的选择 ……（9）
1.3.7　DSP 系统的运算量 ……（10）
1.3.8　DSP 芯片的应用 ……（11）
1.4　小结 ……（12）
1.5　习题与思考题……（12）
第 2 章　DSP 芯片的基本结构和特征……（13）
2.1　引言 ……（13）
2.2　DSP 芯片的基本结构……（13）
2.3　中央处理单元（CPU）……（14）
2.3.1　算术逻辑单元（ALU）……（14）
2.3.2　累加器 ……（15）
2.3.3　桶形移位寄存器……（16）
2.3.4　乘累加单元 ……（16）
2.3.5　寻址单元 ……（18）
2.4　总线结构和流水线 ……（18）
2.5　片内存储器……（20）
2.6　集成外设……（22）
2.6.1　通用 I/O 接口 ……（22）
2.6.2　定时器 ……（22）
2.6.3　主机接口（HPI）……（22）
2.6.4　串行口 ……（23）
2.6.5　软件可编程等待状态
产生器……（24）
2.7　TI 定点 DSP 芯片 ……（24）
2.7.1　早期的定点 DSP 芯片 ……（24）
2.7.2　TMS320C2000 系列 ……（31）
2.7.3　TMS320C5000 系列 ……（32）
2.7.4　TMS320C62x/C64x 系列…（35）
2.8　TI 浮点 DSP 芯片 ……（36）
2.8.1　TMS320C3x……（36）
2.8.2　TMS320C67x……（39）
2.9　其他 DSP 芯片简介 ……（39）
2.9.1　AD 公司的 DSP 芯片 ……（39）
2.9.2　AT&T 公司 DSP 芯片……（41）
2.9.3　Motorola 公司 DSP 芯片 …（41）
2.10　小结……（42）
2.11　习题与思考题……（42）
第 3 章　DSP 的数值运算基础……（43）
3.1　引言……（43）
3.2　定点的基本概念 ……（43）
3.2.1　数的定标……（43）
3.2.2　数的转换……（44）
3.2.3　溢出保护……（45）
3.2.4　符号扩展……（46）
3.2.5　舍入与截尾……（46）
3.3　定点运算实现的基本原理……（47）
3.3.1　加法/减法运算的 C 语言
定点模拟……（47）
3.3.2　乘法运算的 C 语言定点
模拟……（49）
3.3.3　除法运算的 C 语言定点
模拟 ……（50）
3.3.4　程序变量的 Q 值确定……（50）
3.3.5　浮点至定点变换的 C
程序举例……（51）
3.4　DSP 定点算术运算实现的
基本原理……（53）
3.4.1　定点乘法……（53）
3.4.2　定点加法……（55）

3.4.3 定点除法 …… （55）
3.5 非线性运算定点实现方法…… （57）
3.5.1 级数展开法 …… （57）
3.5.2 查表法 …… （58）
3.5.3 混合法 …… （59）
3.6 浮点数的表示格式 …… （61）
3.6.1 IEEE 浮点数格式 …… （61）
3.6.2 TMS320C3x 浮点数格式 ·· （62）
3.7 基本的浮点运算 …… （64）
3.7.1 浮点乘法和加减法 …… （64）
3.7.2 浮点除法 …… （64）
3.8 非线性浮点运算的快速实现…… （65）
3.9 小结 …… （67）
3.10 习题与思考题 …… （67）
第 4 章 DSP 芯片的开发环境 …… （68）
4.1 引言 …… （68）
4.2 软件开发…… （68）
4.3 软件开发环境 …… （70）
4.3.1 CCS 安装与设置 …… （70）
4.3.2 CCS 基本环境 …… （74）
4.3.3 CCS 工程开发实例 …… （88）
4.3.4 DSP/BIOS …… （99）
4.4 硬件开发…… （101）
4.5 硬件开发环境 …… （102）
4.6 小结 …… （103）
4.7 习题与思考题 …… （103）
第 5 章 基于 C 语言的 DSP 芯片开发 …· （104）
5.1 引言 …… （104）
5.2 CCS 支持的 C 语言 …… （104）
5.2.1 变量和常数 …… （105）
5.2.2 函数 …… （110）
5.2.3 预处理 …… （113）
5.3 C 语言程序代码的优化 …… （113）
5.3.1 C 语言程序代码编译分析 …… （113）
5.3.2 C 语言程序的优化方法 … （118）
5.4 小结 …… （122）
5.5 习题与思考题 …… （122）
第 6 章 DSP 芯片的存储资源管理 …… （123）
6.1 引言 …… （123）
6.2 DSP 芯片的存储器…… （123）
6.2.1 片内存储器…… （123）
6.2.2 外部扩展存储器 …… （124）
6.3 存储区的组织 …… （124）
6.3.1 程序空间…… （124）
6.3.2 数据空间…… （125）
6.3.3 I/O 空间 …… （126）
6.3.4 存储器映射寄存器 …… （126）
6.3.5 TMS320VC5416 的存储资源 …… （126）
6.4 DSP 芯片中的代码存储结构…· （128）
6.4.1 块 …… （128）
6.4.2 汇编器对块的处理 …… （129）
6.4.3 链接器对块的处理 …… （130）
6.4.4 程序重定位…… （131）
6.4.5 COFF 文件中的符号 …… （132）
6.4.6 COFF 格式编程示例 …… （133）
6.5 存储区分配示例 …… （134）
6.5.1 TMS320C2xx 链接命令文件示例…… （134）
6.5.2 TMS320VC5416 链接命令文件示例…… （135）
6.6 小结…… （136）
6.7 习题与思考题 …… （136）
第 7 章 TMS320C5000 DSP 芯片的汇编语言…… （137）
7.1 汇编语言源程序格式…… （137）
7.2 汇编源程序中常见符号和伪指令…… （138）
7.3 汇编指令系统 …… （141）
7.4 寻址方式 …… （185）
7.4.1 数据寻址…… （185）
7.4.2 程序寻址…… （194）
7.5 小结…… （196）
7.6 习题与思考题 …… （196）
第 8 章 基于 C 和汇编语言混合编程的 DSP 芯片开发…… （197）
8.1 引言…… （197）
8.2 基于 C 和汇编语言的混合编程方法…… （197）
8.2.1 独立的 C 和汇编模块接口…… （197）

8.2.2 从 C 程序中访问汇编程序变量 …… （200）
8.2.3 在汇编程序中访问 C 程序变量 …… （201）
8.2.4 在 C 程序中直接嵌入汇编语句 …… （202）
8.2.5 修改 C 编译器的输出 …… （204）
8.3 基于 C 和汇编语言的混合编程举例 …… （205）
8.3.1 TMS320C3x/C4x 混合编程举例 …… （205）
8.3.2 TMS320C2x/C5x 混合编程举例 …… （206）
8.3.3 TMS320C54x 混合编程举例 …… （207）
8.4 DSP 代码的优化 …… （207）
8.4.1 C 代码的优化 …… （208）
8.4.2 汇编代码的优化 …… （211）
8.5 小结 …… （218）
8.6 习题与思考题 …… （218）
第 9 章 DSP 芯片集成外设的开发 …… （219）
9.1 引言 …… （219）
9.2 中断 …… （220）
9.2.1 概述 …… （220）
9.2.2 DSP 中断资源 …… （224）
9.2.3 DSP 中断程序实现 …… （226）
9.3 DMA …… （227）
9.3.1 DMA 数据传输模式 …… （228）
9.3.2 DSP 的 DMA 控制 …… （228）
9.3.3 DSP 的 DMA 配置举例 …… （232）
9.4 时序模块 …… （234）
9.4.1 PLL 模块 …… （234）
9.4.2 定时模块 …… （236）
9.4.3 等待状态发生器 …… （238）
9.5 接口模块 …… （240）
9.5.1 概述 …… （240）
9.5.2 多通道缓冲串行口 …… （241）
9.5.3 主机接口（HPI） …… （248）
9.5.4 外部存储器接口（EMIF） …… （253）
9.5.5 通用 I/O 口 …… （258）
9.6 基于 CSL 的外设开发 …… （258）
9.6.1 CSL 概述 …… （259）
9.6.2 CSL 程序开发 …… （259）
9.7 小结 …… （264）
9.8 习题与思考题 …… （264）
第 10 章 DSP 脱机系统设计 …… （265）
10.1 引言 …… （265）
10.2 DSP 芯片的初始化 …… （265）
10.2.1 寄存器初始化 …… （265）
10.2.2 中断矢量表初始化 …… （268）
10.2.3 串行口初始化 …… （270）
10.3 DSP 系统的 BOOT 设计 …… （274）
10.3.1 TMS320C50 的 BOOT 设计 …… （274）
10.3.2 TMS320C54x 的 BOOT 设计 …… （276）
10.3.3 TMS320C55x 的 BOOT 设计 …… （282）
10.3.4 TMS320C6x 的 BOOT 设计 …… （285）
10.4 小结 …… （286）
10.5 习题与思考题 …… （287）
第 11 章 DSP 硬件系统设计 …… （288）
11.1 引言 …… （288）
11.2 DSP 系统的基本硬件设计 …… （288）
11.2.1 电源电路 …… （288）
11.2.2 复位电路 …… （290）
11.2.3 引脚的电平转换与处理 …… （291）
11.2.4 硬件系统的可靠性设计 …… （294）
11.3 DSP 典型硬件系统举例 …… （297）
11.3.1 外部存储器接口设计 …… （297）
11.3.2 模数接口电路的设计 …… （305）
11.3.3 通信接口的设计 …… （309）
11.3.4 主从式系统的设计 …… （311）
11.4 小结 …… （320）
11.5 习题与思考题 …… （320）
第 12 章 DSP 芯片应用系统开发实例 …… （321）
12.1 引言 …… （321）
12.2 一个基于 TMS320C203 DSP

应用系统的开发……（321）
12.2.1 系统简介 ……（321）
12.2.2 系统硬件设计……（322）
12.2.3 系统软件设计……（324）
12.2.4 硬件调试 ……（325）
12.2.5 软件调试 ……（327）
12.2.6 独立系统实现……（327）
12.3 一个基于 TMS320C31 DSP 系统的开发 ……（328）
12.3.1 系统简介 ……（328）
12.3.2 系统构成 ……（328）
12.3.3 系统软硬件设计……（329）
12.3.4 软硬件调试 ……（331）
12.3.5 独立系统实现……（333）
12.4 一个基于 TMS320VC5409 DSP 应用系统的开发 ……（334）
12.4.1 G.729A 及系统简介……（334）
12.4.2 系统构成 ……（335）
12.4.3 系统软硬件设计……（336）
12.4.4 系统调试 ……（337）
12.4.5 独立系统形成……（338）
12.5 小结 ……（340）
12.6 习题与思考题 ……（340）
第 13 章 数字滤波器算法的 DSP 实现…（341）
13.1 引言 ……（341）
13.2 FIR 滤波器的 DSP 实现……（341）
13.2.1 FIR 滤波器的基本原理和设计方法 ……（341）
13.2.2 FIR 滤波器的定点 DSP 实现 ……（344）
13.2.3 FIR 滤波器的浮点 DSP 实现 ……（347）
13.3 IIR 滤波器的 DSP 实现……（349）
13.3.1 IIR 滤波器的基本原理和设计方法 ……（349）
13.3.2 IIR 滤波器的定点 DSP 实现 ……（352）
13.3.3 IIR 滤波器的浮点 DSP 实现……（354）
13.4 自适应滤波器的 DSP 实现……（357）
13.4.1 自适应滤波器的基本原理……（357）
13.4.2 自适应滤波器的 TMS320C2xx 定点实现 ……（358）
13.4.3 自适应滤波器的 TMS320C54x 定点实现 ……（360）
13.4.4 自适应滤波器的 TMS320C3x 浮点实现 ……（364）
13.5 小结……（365）
13.6 习题与思考题 ……（365）
第 14 章 FFT 算法的 DSP 实现 ……（366）
14.1 引言……（366）
14.2 FFT 的基本原理……（366）
14.2.1 DFT 的基本原理……（366）
14.2.2 FFT 算法的导出 ……（367）
14.3 FFT 算法的 C 语言实现 ……（371）
14.4 FFT 的定点 DSP 实现……（373）
14.4.1 运算溢出及避免方法 …（373）
14.4.2 FFT 的 TMS320C2xx 实现……（374）
14.4.3 FFT 的 TMS320C54x 实现……（379）
14.5 FFT 的浮点 DSP 实现……（385）
14.6 小结……（392）
14.7 习题与思考题 ……（392）
附录 A TMS320C54X 指令集 ……（393）
附录 B TI 格式文件转化为二进制文件 ……（400）
附录 C 8 位 μ 律/16 位线性互换的 C 语言程序……（404）
附录 D μ 律到线性变换表……（406）
附录 E 缩写词的英文对照 ……（407）
参考文献 ……（410）

第1章 概 述

1.1 引 言

数字信号处理（DSP）是一门涉及多种学科且又广泛应用于许多领域的学科。20 世纪 60 年代以来，随着信息技术的飞速发展，DSP 技术应运而生并得到迅速的发展。目前，DSP 技术已经在通信、自动控制、航空航天、军事、仪器仪表、家用电器等众多领域里得到越来越广泛的应用，DSP 已经来到了我们每个人的身边。

DSP 是指利用计算机、微处理器或专用处理设备，以数字形式对信号进行的采集、变换、滤波、估值、增强、压缩、识别等处理。

DSP 是围绕着 DSP 的理论、实现和应用等几个方面发展起来的。DSP 在理论上的发展推动了 DSP 应用的发展；反过来，越来越广泛的 DSP 应用又促进了 DSP 理论的发展；DSP 的实现则是理论和应用之间的桥梁。

DSP 以众多学科的理论为基础，它所涉及的范围极其广泛。例如，数学领域的微积分、概率统计、随机过程、数值分析等都是 DSP 的基本工具，与网络理论、信号与系统、控制论、通信理论、故障诊断等也密切相关。一些新兴的学科，如人工智能、模式识别、神经网络等，都是与 DSP 密不可分。因此可以说，DSP 把许多经典的理论体系作为自己的理论基础，同时又使自己成为一系列新兴学科的理论基础。

DSP 的实现一般有以下几种方法：

（1）在通用的计算机（如 PC）上用软件（如 C 语言）实现。

（2）在通用计算机系统中加上专用的加速处理机实现。

（3）用通用的单片机（如 MCS-51、MSP430 系列等）实现。

（4）用通用的可编程 DSP 芯片实现。与单片机相比，DSP 芯片具有更加适合于 DSP 的软件和硬件资源，可用于复杂的 DSP 算法。

（5）用专用的 DSP 芯片实现。在一些特殊的场合，要求的信号处理速度极高，用通用 DSP 芯片很难实现，例如专用于 FFT、数字滤波、卷积、相关等算法的 DSP 芯片，这种芯片将相应的信号处理算法在芯片内部用硬件实现，无需进行编程。

在上述几种方法中，第 1 种方法的优点是实现方便，但缺点是速度较慢，一般可用于 DSP 算法的模拟与仿真；第 2 种方法需要在计算机上增加专用处理板，实现比较复杂，也不便于系统的脱机运行；第 3 种方法只适用于实现简单的 DSP 算法，可用于一些不太复杂的 DSP 应用领域；第 5 种方法专用性强，仅适用于特定的算法，应用受到比较大的限制；而第 4 种方法，由于 DSP 芯片的软硬件资源丰富，开发也比较方便，为 DSP 的应用打开了新的局面。

虽然 DSP 的理论发展迅速，但在 20 世纪 80 年代以前，由于实现方法的限制，DSP 的理论还得不到广泛的应用。直到 20 世纪 80 年代初世界上第一片单片可编程 DSP 芯片的诞生，才将理论研究成果广泛应用到低成本的实际系统中，并且推动了新的理论和应用领域的发展。

可以毫不夸张地说，DSP 芯片的诞生及发展对最近二十多年来自动控制、通信、计算机应用等领域的发展起到十分重要的作用。

学习掌握 DSP 芯片的开发应用对于实现各种 DSP 的应用系统具有十分重要的实际意义。

1.2 DSP 系统

1.2.1 DSP 系统的基本构成

图 1-1 示出了一个典型的 DSP 系统。图中，输入信号可以有各种各样的形式，例如，可以是麦克风采集的语音信号，也可以是编码后在数字链路上的传输信号或存储在计算机里的摄像机图像信号等。

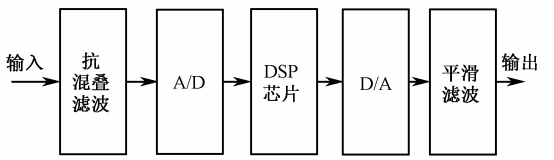


图 1-1 典型的 DSP 系统

一般地，输入信号首先进行带限滤波和抽样，然后进行模/数（A/D）变换将信号变换成数字比特流。根据奈奎斯特抽样定理，对低通模拟信号，为保持信息的不丢失，抽样频率至少必须是输入带限信号最高频率的 2 倍。

DSP 芯片的输入是 A/D 变换后得到的时间离散的数字信号，根据系统要求，DSP 芯片对输入信号按照特定的算法进行处理，这是 DSP 系统的关键。最后，处理后的数字样值再经数/模（D/A）变换转换为模拟样值，之后再进行平滑滤波就可得到连续的模拟波形。

需要指出的是，上面给出的 DSP 应用系统只是一个典型的模型系统，并不是所有的 DSP 应用系统都必须包括模型系统中的各个部分。例如，语音识别系统在输出端并不是连续的波形，而是识别结果，如文字、数字、标点符号等。有些输入信号本身就是数字信号，如 CD 信号，因此就不必进行 A/D 变换了。

1.2.2 DSP 系统的特点

DSP 系统以数字信号处理为基础，因此具有数字处理的全部优点：

（1）接口方便。DSP 应用系统与其他以现代数字技术为基础的系统或设备都是相互兼容的。与这样的系统接口以实现某种功能要比模拟系统与这些系统接口要容易得多。

（2）编程方便。DSP 应用系统中的可编程 DSP 芯片可使设计人员在开发过程中灵活方便地对软件进行修改和升级。

（3）稳定性好。DSP 应用系统以数字处理为基础，受环境温度以及噪声的影响较小，可靠性高。

（4）精度高。16 位数字系统可以达到 10^{-5} 的精度（$1/2^{15}$）。

（5）可重复性好。模拟系统的性能受元器件参数性能变化比较大，而数字系统基本不受影响，因此数字系统便于调试、测试和大规模生产。

（6）集成方便。DSP 应用系统中的数字部件有高度的规范性，便于大规模集成。

当然，数字信号处理也存在一定的缺点。例如，对于简单的信号处理任务，若采用 DSP 则可能会使成本增加；DSP 系统中的高速时钟可能带来高频干扰和电磁泄漏等问题。

DSP 应用系统的突出优点已经使之在工业控制、通信、语音、图像、雷达、生物医学、仪器仪表等许多领域得到越来越多的应用。

1.2.3　DSP 系统的设计

图 1-2 是 DSP 应用系统设计的一般流程。

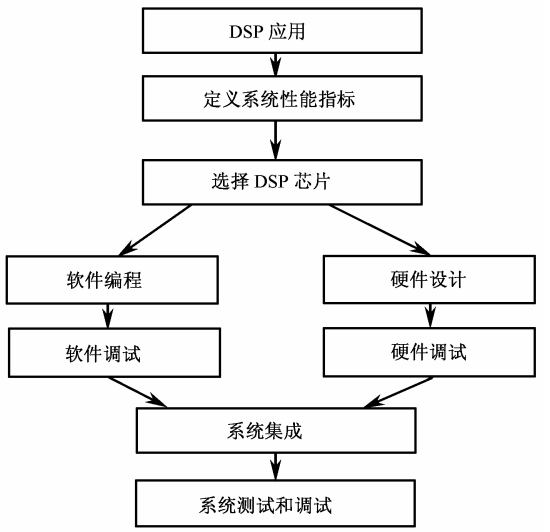


图 1-2　DSP 应用系统设计的一般流程

1．定义系统性能指标

在设计 DSP 系统之前，首先必须根据应用系统的目标确定系统的性能指标、信号处理的要求。通常可用数据流程图、数学运算序列、正式的符号或自然语言来描述。

2．采用高级语言进行性能模拟

一般来说，为了实现系统的最终目标，需要对输入的信号进行适当的处理，而处理方法的不同会导致不同的系统性能。要得到最佳的系统性能必须在这一步确定最佳的处理方法，即数字信号处理的算法，因此这一步也称算法模拟或仿真阶段。例如，语音编码算法就是要在确定的压缩比条件下，获得最佳的合成语音。算法模拟所用的输入数据是实际采集信号，通常以计算机文件的形式存储为数据文件。有些算法模拟时所用的输入数据并不一定要是实际采集的信号数据，只要能够验证算法的可行性，输入假设的数据也是可以的。

3．设计实时DSP应用系统

实时 DSP 应用系统的设计包括硬件设计和软件设计两个方面。硬件设计首先要根据系统运算量的大小、对运算精度的要求、系统成本限制以及体积、功耗等要求选择合适的 DSP 芯片。然后，设计 DSP 芯片的外围电路及其他电路。软件设计和编程主要根据系统要求和所选的 DSP 芯片编写相应的 DSP 程序，可以采用汇编语言，也可以用高级语言（如 C 语言）编程。当然由

于高级语言并不是完全针对数字信号处理应用设计的，而基于芯片结构设计的汇编语言在实现代码时的效率更高，因此在实际应用系统中常常采用高级语言和汇编语言的混合编程方法，即首先利用高级语言进行编程，然后针对算法运算量大的地方，利用手工的方式进行优化。采用这种方法，既可缩短软件开发的周期，提高程序的可读性和可移植性，又能满足系统实时运算的要求。

4．借助开发工具进行软硬件调试

软件的调试一般借助于 DSP 开发工具，如软件模拟器（Simulator）、DSP 开发系统或仿真器（Emulator）等。调试 DSP 算法时，一般采用将实时结果与模拟结果进行比较的方法，如果实时程序和模拟程序的输入相同，则两者的输出应该一致。应用系统的其他软件可以根据实际情况进行调试。硬件调试一般采用硬件仿真器进行调试，如果没有相应的硬件仿真器，且硬件系统不是十分复杂，也可以借助于一般的工具进行调试。

5．系统集成与独立系统运行

系统的软件和硬件分别调试完成后，就可以将软件脱离开发系统而直接在应用系统上运行。当然，DSP 应用系统的开发，特别是软件开发是一个需要反复进行的过程，虽然通过算法模拟基本上可以知道实时系统的性能，但实际上模拟环境不可能做到与实时系统环境完全一致，而且将模拟算法移植到实时系统时必须考虑算法是否能够实时运行的问题。如果算法运算量太大不能在硬件上实时运行，则必须重新修改或简化算法。



1.2.4　DSP 系统的开发

根据图 1-2 的设计流程，要开发一个完整的 DSP 应用系统，需要借助于诸多软硬件开发工具，表 1-1 列出了可能需要的开发工具。需要注意的是，有些工具也不一定是必备的，如逻辑分析仪。有些工具则是可选的，如算法模拟时可以用 C 语言，也可以用 MATLAB 语言，或者先进行 MATLAB 模拟，再进行 C 语言模拟，还可以用其他程序语言。在采用美国得克萨斯仪器公司（Texas Instruments， TI）的 DSP 芯片进行系统开发时，一般需采用 CCS（Code Composer Studio）工具软件，这是一个集成开发环境，包括了编辑、编译、汇编、链接、软件模拟、调试等几乎所有需要的软件。此外，如果 DSP 应用系统中还有其他微处理器（如 MCS-51 系列单片机），当然还必须有相应的开发工具支持。

表 1-1　DSP 应用系统开发工具支持

开发步骤	开　发 内　容	开 发 工 具 支 持	
		硬件支持	软件支持
1	算法模拟	计算机	C 语言、MATLAB 语言等
2	DSP 软件编程	计算机	编辑器（如 Edit、Ultraedit 等）
3	DSP 软件调试	计算机、DSP 仿真器等	DSP 代码生成工具（包括 C 编译器、汇编器、链接器等）、DSP 软件模拟器 Simulator、CCS 等
4	DSP 硬件设计	计算机	电路设计软件（如 Protel 99 等）、 其他相关软件（如 EDA 软件等）
5	DSP 硬件调试	计算机、DSP 仿真器、示波器、信号发生器、逻辑分析仪等	相关支持软件
6	系统集成	计算机、DSP 仿真器、编程器、示波器、信号发生器、逻辑分析仪等	相关支持软件

1.3 DSP 芯片

1.3.1 DSP 芯片的定义

DSP 芯片，即数字信号处理芯片（也称数字信号处理器），是一种特别适合于进行 DSP 的微处理器，其主要应用是实时快速地实现各种 DSP 算法。根据 DSP 的要求，DSP 芯片一般具有如下一些优点：

（1）在一个指令周期内一般至少可完成一次乘法和一次加法；
（2）程序空间和数据空间分开，可以同时访问指令和数据；
（3）片内具有快速 RAM，通常可通过独立的数据总线在两块中同时访问；
（4）具有低开销或无开销循环及跳转的硬件支持；
（5）快速的中断处理和硬件 I/O 支持；
（6）具有在单周期内操作的多个硬件地址产生器；
（7）可以并行执行多个操作；
（8）支持流水线操作，使取指、译码和执行等操作可以并行执行。

与通用微处理器相比，早期推出的 DSP 芯片的其他通用功能相对较弱些。但是，近些年来新推出的 DSP 芯片已经将通用微处理器的一些功能集成在芯片中，有的甚至将 DSP 核和 ARM 核集成在一个芯片中（如 OMAP），DSP 芯片已经可以实现普通微处理器的功能。目前市场上推出的智能手机，大多是基于 OMAP 平台实现的。

1.3.2 DSP 芯片的特点

为了快速地实现 DSP 运算，DSP 芯片一般都采用特殊的软硬件结构和指令系统。下面以美国 TI 公司的 TMS320 系列为例介绍 DSP 芯片的主要特点。

TMS320 系列 DSP 芯片的主要特点包括：（1）采用哈佛结构实现内部总线；（2）采用流水线操作方式实现指令操作；（3）乘法采用专用的硬件乘法器实现；（4）具有高效的 DSP 指令。这些特点使得 DSP 芯片可以实现快速的 DSP 运算，并使大部分运算（例如乘累加操作）能够在一个指令周期内完成。由于 TMS320 DSP 芯片是软件可编程器件，因此具有通用微处理器方便灵活的特点。

1．哈佛结构

哈佛结构的主要特点是将程序和数据存储在不同的存储空间中，即程序存储器和数据存储器是两个相互独立的存储器，每个存储器独立编址、独立访问。与两个存储器相对应的是系统中设置了程序总线和数据总线两条总线，从而使数据的吞吐率提高了一倍。

2．流水线操作

与哈佛结构相关，DSP 芯片广泛采用流水线以减少指令执行时间，从而增强了处理器的处理能力。采用流水线操作，处理器可以并行处理多条指令，每条指令处于流水线上的不同阶段。

3．专用的硬件乘法器

乘法是 DSP 的重要组成部分。例如，在一般形式的 FIR 滤波器中，对每个滤波器抽头，必须做一次乘法和一次加法。乘法速度越快，DSP 芯片的性能就越高。在早期的通用微处理器中，乘法指令由一系列加法来实现，故需许多个指令周期来完成。相比而言，DSP 芯片的特征就是有一个专用的硬件乘法器。在 TMS320 系列中，由于具有专用的硬件乘法器，乘法可在一个指令周期内完成。从最早的 TMS32010 实现 FIR 的每个抽头算法可以看出，滤波器每个抽头需要一条乘法指令 MPY：

```
LT      ；装乘数到 T 寄存器
DMOV    ；在存储器中移动数据以实现延迟
MPY     ；相乘
APAC    ；将乘法结果加到 ACC 中
```

其他三条指令用来将乘数装入到乘法器电路（LT）、移动数据（DMOV）以及将乘法结果（存在乘积寄存器 P 中）加到 ACC 中（APAC）。因此，若采用 256 抽头的 FIR 滤波器，这四条指令必须重复执行 256 次，且 256 次乘法必须在一个抽样间隔内完成。在典型的通用微处理器中，每个抽头需要 30～40 个指令周期，而 TMS32010 只需 4 条指令。如果采用特殊的 DSP 指令或采用 TMS320C54x 等新一代的 DSP 芯片，可进一步降低 FIR 抽头的计算时间。

4．高效的DSP指令

DSP 芯片的另一个特征是采用高效的指令。例如，在数字信号处理中，延迟操作非常重要，在 TMS32010 中这个延迟就是由 DMOV 来实现，它完成数据移位功能。TMS32010 中的另一个高效指令是 LTD，它在一个指令周期内完成 LT、DMOV 和 APAC 三条指令。LTD 和 MPY 指令可以将 FIR 滤波器抽头计算从 4 条降为 2 条。在第二代处理器中，如 TMS320C25，增加了一些更高效的指令，即 RPT（RPTK）和 MACD 指令，采用这些高效指令，可以进一步将每个抽头的运算时间从 2 条降为 1 条：

```
RPTK    255     ；重复执行下条指令 256 次
MACD            ；可完成 LT、DMOV、MPY 及 APAC 指令功能
```

哈佛结构、流水线操作、专用的硬件乘法器、高效的 DSP 指令再加上集成电路的优化设计可使 DSP 芯片的指令周期在 200 ns 以下。TMS320 系列 DSP 芯片的指令周期已经从第一代的 400 ns 降低至现在的 20 ns 以下。快速的指令执行时间使得 DSP 芯片能够实时实现许多 DSP 应用。

1.3.3 DSP 芯片的发展

世界上第一片单片 DSP 芯片应当是 1978 年 AMI 公司宣布的 S2811，1979 年美国 Intel 公司宣布的商用可编程器件 2920 是 DSP 芯片的一个主要里程碑。但这两种芯片内部都没有现代 DSP 芯片所必须具有的单周期乘法器。1980 年，日本 NEC 公司推出的 uPD7720 是第一片具有乘法器的商用 DSP 芯片。

在这之后，最成功的 DSP 芯片当数美国 TI 公司的一系列产品。TI 公司在 1982 年成功推出其第一代 DSP 芯片 TMS32010 及其系列产品 TMS32011、TMS320C10/C14/C15/C16/C17 等，之后相继推出了第二代 DSP 芯片 TMS32020、TMS320C25/C26/C28，第三代 DSP 芯片 TMS320C30/C31/C32，第四代 DSP 芯片 TMS320C40/C44，第五代 DSP 芯片 TMS320C5x/C54x，

第二代 DSP 芯片的改进型 TMS320C2xx，集多片 DSP 芯片于一体的高性能 DSP 芯片 TMS320C8x 以及目前速度最快的第六代 DSP 芯片 TMS320C62x/C64x/C67x 等。

第一片采用 CMOS 工艺生产浮点 DSP 芯片的是日本的 Hitachi 公司，该公司于 1982 年推出了浮点 DSP 芯片。1983 年，日本 Fujitsu 公司推出的 MB8764，其指令周期为 120 ns，且具有双内部总线，从而使处理的吞吐量发生了一个大的飞跃。而第一片高性能的浮点 DSP 芯片应是 AT&T 公司于 1984 年推出的 DSP32。

与其他公司相比，Motorola 公司在推出 DSP 芯片方面相对较晚。1986 年，该公司推出了定点处理器 MC56001。1990 年，推出了与 IEEE 浮点格式兼容的浮点 DSP 芯片 MC96002。

美国模拟器件公司（Analog Devices，简称 AD 公司）在 DSP 芯片市场上也占有较大的份额，相继推出了一系列具有自己特点的 DSP 芯片，其定点 DSP 芯片有 ADSP2101/2103/2105、ADSP2111/2115、ADSP2161/2162/2164 以、ADSP2171/2181 等，浮点 DSP 芯片有 ADSP21000/21020、ADSP21060/21062 等。

自 1980 年以来，DSP 芯片得到了突飞猛进的发展，DSP 芯片的应用越来越广泛。从运算速度来看，MAC（一次乘法和一次加法）时间已经从 20 世纪 80 年代初的 400 ns（如 TMS32010）降低到 10 ns 以下（如 TMS320C54x/C55x、TMS320C62x/67x/C64x 等），处理能力提高了几十倍，甚至上百倍，DSP 芯片片内 RAM 数量增加一个数量级以上。从制造工艺来看，1980 年采用 4 μm 的 N 沟道 MOS（NMOS）工艺，而现在则普遍采用亚微米 CMOS 工艺。DSP 芯片的引脚数量从 1980 年的最多 64 个增加到现在的 200 个以上，引脚数量的增加，意味着结构灵活性的增加，如外部存储器的扩展和处理器间的通信等。芯片封装也从开始的 DIP 封装变成现在的 BGA 封装。此外，DSP 芯片的发展，使 DSP 应用系统的成本、体积、重量和功耗都有很大程度的下降。表 1-2 是 TI 公司 DSP 芯片 1982 年、1992 年、1999 年、2002 年的典型值发展比较表。

表 1-2 TI DSP 芯片发展比较表（典型值）

年份	1982	1992	1999	2002
制造工艺	4 μm NMOS	0.8 μm CMOS	0.3 μm CMOS	<0.18 μm CMOS
MIPS	5 MIPS	40 MIPS	100 MIPS	160 MIPS
MHz	20 MHz	80 MHz	100 MHz	160 MHz
内部 RAM	144 字	1 K 字	32 K 字	64 K 字
内部 ROM	1.5 K 字	4 K 字	16 K 字	32 K 字
价格	$150.00	$15.00	$5.00~$25.00	$5.00~$25.00
功耗	250 mW/MIPS	12.5 mW/MIPS	0.45 mW/MIPS	0.05 mW/MIPS



1.3.4 TI 公司的 DSP 芯片

目前，TI 公司的一系列 DSP 产品已经成为当今世界上最有影响的 DSP 芯片。TI 公司已经成为世界上最大的 DSP 芯片供应商，其 DSP 市场占全世界份额的近 50%。

TI 公司常用的 DSP 芯片，可以归纳为三大系列，即

- TMS320C2000 系列：包括 TMS320C2xx/C24x/C28x 等；
- TMS320C5000 系列：包括 TMS320C54x/C55x/OMAP 等；
- TMS320C6000 系列：包括 TMS320C62x/C67x/C64x 等。

此外，还有浮点芯片 TMS320C3x 系列，其典型芯片是 TMS320VC33。

TMS320C2000 系列主要是定点 DSP 芯片，主要面向自动控制领域，目前主推 TMS320C24x 和 TMS320C28x。TMS320C24x 提供 20~40 MIPS 的运算速度，内部集成了如 PWM、A/D 转换等许多工业控制领域所需的资源，性能价格比较高。TMS320C28x 是近年推出的新一代工业控制芯片，可提供 150 MIPS 的运算速度，采用 32 位 CPU，片内的资源丰富，可以单片实现大部分应用系统。值得注意的是，该系列芯片目前也有浮点芯片，如 TMS320F283x，该芯片采用 32 位浮点 C28x 内核，运算速度可达 300 MFLOPS。

TMS320C5000 系列是定点 DSP 芯片，主要面向通信、信息技术领域，目前主推 TMS320C54x、TMS320C55x 和 OMAP。TMS320C54x 系列成员众多，可提供各种性能选择，最高速度可达几百 MIPS。TMS320C55x 在 TMS320C54x 的基础上采用高性能的电源管理技术，成为目前功耗最低的一类 DSP 芯片，特别适用于需要电池供电的应用场合。OMAP 芯片集 TMS320C55x 与 ARM 处理器于一体，可在很多应用场合真正做到单片实现。

TMS320C6000 系列应该是目前 TI DSP 芯片的顶级产品，具有最高的性能，且便于高级语言编程。特别适用于需要高性能处理的场合，如图像和视频处理等。该系列既有定点芯片（TMS320C62x/C64x），也有浮点芯片（TMS320C67x）。其中，TMS320C64x 的时钟速度可达 1GHz，TMS320C67x 在提供高的运算精度的同时，运算速度达 600~1800 MFLOPS。



1.3.5 DSP 芯片的分类

DSP 芯片可以按照下列三种方式进行分类。

1．按基础特性分

这是根据 DSP 芯片的工作时钟和指令类型来分类的。如果在某时钟频率范围内的任何时钟频率上，DSP 芯片都能正常工作，除计算速度有变化外，没有性能的下降，这类 DSP 芯片一般称之为静态 DSP 芯片。例如，日本 OKI 电气公司的 DSP 芯片、TI 公司的 TMS320 系列芯片属于这一类。

如果有两种或两种以上的 DSP 芯片，它们的指令集和相应的机器代码机引脚结构相互兼容，则这类 DSP 芯片称为一致性 DSP 芯片。例如，美国 TI 公司的 TMS320C54x 就属于这一类。

2．按数据格式分

这是根据 DSP 芯片工作的数据格式来分类的。数据以定点格式工作的 DSP 芯片称为定点 DSP 芯片，如 TI 公司早期推出的 TMS320C1x/C2x/C5x、TMS320C24x、TMS320C5000 系列，TMS320C6000 系列中的 TMS320C62x/C64x，AD 公司的 ADSP21xx 系列，AT&T 公司的 DSP16/16A，Motorola 公司的 MC56000 等。

以浮点格式工作的称为浮点 DSP 芯片，如 TI 公司的 TMS320C3x/C4x/C67x，AD 公司的 ADSP21xxx 系列，AT&T 公司的 DSP32/32C，Motorola 公司的 MC96002 等。

不同浮点 DSP 芯片所采用的浮点格式不完全一样，有的 DSP 芯片采用自定义的浮点格式，如 TMS320C3x，而有的 DSP 芯片则采用 IEEE 的标准浮点格式，如 Motorola 公司的 MC96002、Fujitsu 公司的 MB86232 和 Zoran 公司的 ZR35325 等。

3．按用途分

按照 DSP 的用途来分，可分为通用型 DSP 芯片和专用型 DSP 芯片。通用型 DSP 芯片适

合普通的 DSP 应用，如 TI 公司的一系列 DSP 芯片属于通用型 DSP 芯片。专用 DSP 芯片是为特定的 DSP 运算而设计，更适合特殊的运算，如数字滤波、卷积和 FFT，如 Motorola 公司的 DSP56200、Zoran 公司的 ZR34881、Inmos 公司的 IMSA100 等就属于专用型的 DSP 芯片。

本书主要讨论通用型的 DSP 芯片。



1.3.6 DSP 芯片的选择

设计 DSP 应用系统，选择 DSP 芯片是非常重要的一个环节。只有选定了 DSP 芯片才能进一步设计其外围电路及系统的其他电路。早期由于 DSP 芯片的种类很少，几乎没有可以选择的余地，只能根据一种或几种 DSP 芯片来设计应用系统。由于 DSP 发展速度很快，现在 DSP 芯片的选择余地很大。总的来说，现在完全可以根据实际应用系统的需要来选择 DSP 芯片，以达到系统的最优化设计。不同的 DSP 应用系统由于应用场合、应用目的等不尽相同，对 DSP 芯片的选择也是不同的。一般来说，选择 DSP 芯片时应考虑到如下诸多因素。

1．DSP芯片的运算速度

运算速度是 DSP 芯片的一个最重要的性能指标，也是选择 DSP 芯片时所需要考虑的一个主要因素。DSP 芯片的运算速度可以用以下几种性能指标来衡量：

（1）指令周期：即执行一条指令所需的时间，通常以 ns（纳秒）为单位。如 TMS320VC5402-100 在主频为 100 MHz 时的指令周期为 10 ns。

（2）MAC 时间：即完成一次乘法和一次加法的时间。大部分 DSP 芯片可在一个指令周期内完成一次乘法和一次加法操作，如 TMS320VC5402-100 的 MAC 时间就是 10 ns。

（3）FFT 执行时间：即运行一个 N 点 FFT 程序所需的时间。由于 FFT 运算涉及的运算在数字信号处理中很有代表性，因此 FFT 运算时间常作为衡量 DSP 芯片运算能力的一个指标。

（4）MIPS：即每秒执行百万条指令。如 TMS320VC5402-100 的处理能力为 100 MIPS，即每秒可执行 1 亿条指令。

（5）MOPS：即每秒执行百万次操作。如 TMS320C40 的运算能力为 275 MOPS。

（6）MFLOPS：即每秒执行百万次浮点操作。如 TMS320C31 在主频为 40MHz 时的处理能力为 40 MFLOPS。

（7）BOPS：即每秒执行十亿次操作。如 TMS320C80 的处理能力为 2 BOPS。

2．DSP芯片的价格

DSP 芯片的价格也是选择 DSP 芯片所需考虑的一个重要因素。如果采用价格昂贵的 DSP 芯片，即使性能再高，其应用范围肯定会受到一定的限制，尤其是需要大规模推广应用的民用产品。因此，根据实际系统的应用情况，需要确定一个价格适中的 DSP 芯片。当然，由于 DSP 芯片发展速度很快，DSP 芯片的价格往往下降也较快，因此在开发阶段选用某种价格稍贵的 DSP 芯片，等到系统开发完毕，其价格可能已经下降一倍甚至更多。

3．DSP芯片的硬件资源

不同的 DSP 芯片所提供的硬件资源是不相同的，如片内 RAM、ROM 的数量，外部可扩展的程序和数据空间，总线接口、I/O 接口等。即使是同一系列的 DSP 芯片（如 TI 的 TMS320C54x 系列），系列中不同 DSP 芯片也具有不同的内部硬件资源，可以满足不同应用的需要。

4．DSP芯片的运算精度

一般的定点 DSP 芯片的字长为 16 位，如 TMS320 系列。但有的公司的定点芯片为 24 位，如 Motorola 公司的 MC56001 等。浮点芯片的字长一般为 32 位，累加器为 40 位。

5．DSP芯片的开发工具

在 DSP 系统的开发过程中，开发工具是必不可少的。如果没有开发工具的支持，要想开发一个复杂的 DSP 系统几乎是不可能的。如果有功能强大的开发工具的支持（如可支持 C 语言开发），则程序开发的时间就会大大缩短。所以，在选择 DSP 芯片的同时，必须注意其开发工具的支持情况，包括软件和硬件的开发工具。

6．DSP芯片的功耗

在某些 DSP 应用场合，功耗也是一个需要特别关注的问题。如手持式设备、便携式设备、野外应用的设备等对功耗都有特殊的要求。目前，3.3 V 及以下供电的低功耗高速 DSP 芯片已大量使用。

7．其他因素

除了上述因素外，选择 DSP 芯片还应考虑到封装的形式、质量标准、供货情况、生命周期等。有的 DSP 芯片可能有 DIP、PGA、PLCC、PQFP、BGA 等多种封装形式。有些 DSP 系统可能最终要求的是工业级或军用级标准，在选择时就需要注意到所选的芯片是否有工业级或军用级的同类产品。如果所设计的 DSP 系统不仅仅是一个实验系统，而是需要批量生产并可能有几年甚至十几年的生命周期，那么需要考虑所选的 DSP 芯片供货情况如何，是否也有同样甚至更长的生命周期等。

在上述诸多因素中，一般而言，定点 DSP 芯片的价格较为便宜、功耗较低，但运算精度稍低。而浮点 DSP 芯片的优点是运算精度高、且 C 语言编程调试方便，但价格稍贵，功耗也较大。例如 TI 的 TMS320C24x 系列属于定点 DSP 芯片，低功耗和低成本是其主要的特点。而 TMS320C3x/C4x/C67x 属于浮点 DSP 芯片，运算精度高，用 C 语言编程方便，开发周期短，但同时其价格和功耗也相对较高。

 1.3.7　DSP 系统的运算量

DSP 应用系统的运算量是确定选用具有恰当处理能力 DSP 芯片的基础。运算量小则可以选用处理能力不是很强的 DSP 芯片，从而可以降低系统成本；相反，运算量大的 DSP 系统则必须选用处理能力强的 DSP 芯片。如果单个 DSP 芯片的处理能力达不到系统要求，则必须采用多个 DSP 芯片进行并行处理。

那么，如何确定DSP系统的运算量以选择DSP芯片呢？下面我们来考虑两种情况。

1．按样点处理

所谓按样点处理就是 DSP 算法对每一个输入样点循环一次。数字滤波就是这种情况。在数字滤波器中，通常需要对每一个输入样点计算一次。例如，一个采用 LMS 算法的 256 抽头的自适应 FIR 滤波器，假定每个抽头的计算需要 3 个 MAC 周期，则 256 抽头计算需要 256×3＝768 个 MAC 周期。如果采样频率为 8 kHz，即样点之间的间隔为 125 μs，DSP 芯片的 MAC 周期为 200 ns，则 768 个 MAC 周期需要 153.6 μs 的时间，显然无法实时处理，需要选用速度更高的 DSP

芯片。表 1-3 示出了两种信号带宽对 DSP 芯片的处理要求，三种 DSP 芯片的 MAC 周期分别为 200 ns、50 ns 和 25 ns。从表中可以看出，对语音应用，后两种 DSP 芯片可以实时实现，对音频应用，只有第三种 DSP 芯片能够实时处理。当然，在这个例子中，没有考虑其他的运算量。

表 1-3 用 DSP 芯片实现数字滤波

应用领域	采样率（kHz）	采样周期（μs）	256 抽头 LMS 滤波运算量（MAC 数）	每样点允许 MAC 指令数（200 ns）	每样点允许 MAC 指令数（50 ns）	每样点允许 MAC 指令数（25 ns）
语音	8	125	768	625	2500	5000
音频	44.1	22.7	768	113	453	907

2．按帧处理

有些数字信号处理算法不是每个输入样点循环一次，而是每隔一定的时间间隔（通常称为帧）循环一次。例如，中低速语音编码算法通常以 10 ms 或 20 ms 为一帧，即每隔 10 ms 或 20 ms 语音编码算法循环一次。所以，选择 DSP 芯片时应该比较一帧内 DSP 芯片的处理能力和 DSP 算法的运算量。

假设 DSP 芯片的指令周期为 p（ns），一帧的时间为Δτ（ns），则该 DSP 芯片在一帧内所能提供的最大运算量为Δτ/p 条指令。例如，设 TMS320VC5402 的指令周期为 10 ns，帧长为 20 ms，则一帧内 TMS320VC5402 所能提供的最大运算量为 20 ms/10 ns＝200 万条指令。因此，理论上只要语音编码算法的运算量不超过 200 万条指令，就可以在 TMS320VC5402 上实时运行。当然，在实际系统中，需要留有一定的余量，以确保程序运行的可靠性。



1.3.8 DSP 芯片的应用

自从 20 世纪 80 年代初 DSP 芯片诞生以来，DSP 芯片得到了飞速的发展。DSP 芯片的高速发展，一方面得益于集成电路技术的发展，另一方面也得益于巨大的应用市场。在 20 多年时间里，DSP 芯片已经在通信、信号处理、自动控制、雷达等许多领域得到广泛的应用。目前，DSP 芯片的价格越来越低，性能价格比日益提高，具有广阔的应用前景。

DSP 芯片的应用领域主要有：

（1）信号处理——如数字滤波、自适应滤波、快速傅里叶变换、相关运算、谱分析、卷积、模式识别、波形产生等；

（2）自动控制——如工业控制、引擎控制、声控、自动驾驶、机器人控制、磁盘控制等；

（3）通信——如调制解调、自适应均衡、数据加密、数据压缩、回波抵消、多路复用、传真、扩频通信、纠错编码、可视电话等；

（4）语音——如语音编码、语音合成、语音识别、语音增强、说话人辨认、说话人确认、语音邮件、语音存储等；

（5）图形/图像——如二维和三维图形处理、图像压缩与传输、图像增强、动画、机器人视觉等；

（6）军事——如保密通信、雷达处理、声纳处理、导航、导弹制导等；

（7）仪器仪表——如频谱分析、函数产生、锁相环等；

（8）医疗——如助听、超声设备、诊断工具、病人监护等；

（9）家用电器——如高保真音响、音乐合成、音调控制、玩具与游戏、数字电话/电视等。

随着 DSP 芯片性能价格比的不断提高，DSP 芯片将会在更多的领域内得到更为广泛的应用。

1.4 小　　结

本章介绍了 DSP 芯片的基本概念，以及 DSP 系统的构成、特点、设计过程和开发方法，重点对 DSP 芯片的一些基础知识进行了阐述，学习掌握这些基础知识可以为后续知识的学习奠定基础。由于 DSP 芯片种类较多，在一本书内包罗万象不仅没有可能，而且也毫无必要。实际上，各种 DSP 芯片的结构和开发方法基本类似，因此在开始学习 DSP 芯片时，可将某类 DSP 芯片作为重点。

本书的原理与方法适用于所有 DSP 芯片。在举例说明时，主要以 TMS320C5000 中的 TMS320C54x/C55x 为主。

1.5 习题与思考题

（1）DSP 算法的实现方法主要有哪些？

（2）简述 DSP 应用系统的典型构成和特点。

（3）简述 DSP 应用系统的一般设计过程。

（4）开发 DSP 应用系统，一般需要哪些软硬件工具？

（5）什么是可编程 DSP 芯片？它有什么特点？

（6）简述 DSP 芯片的发展历程。

（7）什么是定点 DSP 芯片和浮点 DSP 芯片？各有什么优缺点？

（8）设计 DSP 应用系统时，如何选择合适的 DSP 芯片？

（9）TMS320VC5416-160 工作在 160 MHz 时的指令周期是多少 ns？它的运算速度是多少 MIPS？当工作在 100MHz 时，其指令周期和运算速度又是多少？

（10）一个 DSP 系统的采样频率是 10 kHz，采用的 DSP 芯片的指令周期是 10 ns。如果某 DSP 算法是按样点处理的，问算法实时运行的条件是什么？如果 DSP 算法是按帧处理的，且帧长是 10 ms，则在一帧时间内最多可运行多少个指令周期？

（11）写出以下缩写词的中英文全称：DSP、TI、MAC、MIPS、MOPS、MFLOPS、BOPS、DIP、PGA、PLCC、PQFP、PWM。

第 2 章　DSP 芯片的基本结构和特征

2.1　引　　言

可编程 DSP 芯片是一种具有特殊结构的微处理器，为了达到快速进行数字信号处理的目的，DSP 芯片一般都具有程序和数据分开的总线结构、流水线操作功能、单周期完成乘法的硬件乘法器以及一套适合数字信号处理的指令集。

在 DSP 芯片应用中，许多特殊功能是与 DSP 芯片内部的特殊结构紧密相关的。因此，学习 DSP 芯片的结构和特征，对于深入理解 DSP 芯片的操作过程，掌握 DSP 芯片的开发和应用技术具有很重要的意义。

本章首先介绍 DSP 芯片的基本结构，随后重点介绍 TI 公司 DSP 芯片的特点，最后简要介绍其他公司的 DSP 芯片的特点。

2.2　DSP 芯片的基本结构

DSP 芯片的基本结构大致可以分为 CPU、总线、存储器以及集成外设与专用硬件电路等部分。CPU 主要包括算术逻辑单元（ALU）、累加器（ACC）、乘累加单元（MAC）、移位寄存器和寻址单元等。存储器包括片内 ROM、Flash、单访问 RAM（SARAM）、双访问 RAM（DARAM）等。集成外设与专用硬件电路包括片内各种类型的串行接口、主机接口、定时器、时钟发生器、锁相环以及各种控制电路。总线在 CPU 与存储器、集成外设和专用硬件电路等部分之间传送指令和数据，起到桥梁的作用。图 2-1 给出了一个具有典型特点的 DSP 芯片结构框图。

为了适应数字信号处理应用的不同需要，DSP 芯片一般都采用了一些特殊的软硬件结构。下面我们以 TI 公司的 DSP 芯片为例介绍 DSP 芯片的基本结构。

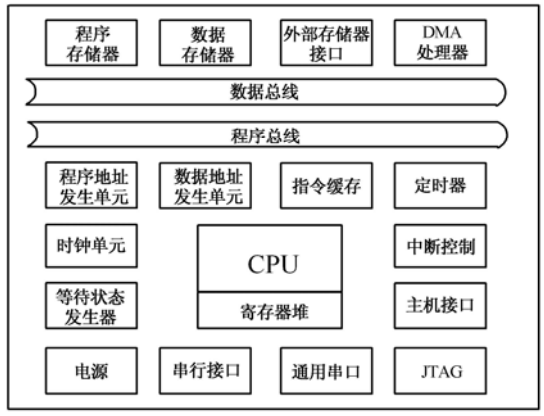


图 2-1　具有典型特点的 DSP 芯片结构

TI 公司的 TMS320 系列 DSP 芯片种类很多，各芯片体系的硬件结构并不完全一致。但是，同一系列的不同型号 DSP 芯片之间，其 CPU 结构与功能基本相同，型号之间的差异主要体现在片内的存储器容量、集成外设、供电电压、速度以及封装上。当然，不同系列 DSP 芯片的最基本硬件操作单元仍是相同或相近的，如硬件乘法器、ALU 单元、桶形移位器等。

2.3 中央处理单元（CPU）

TMS320 系列 DSP 芯片的 CPU 主要由以下几个部分组成：

- 指令解码部分；
- 运算与逻辑部分；
- 寻址部分。

其中，运算与逻辑部分一般包括以下单元：

- 算术逻辑单元（ALU）；
- 累加器 ACC；
- 桶形移位寄存器；
- 乘累加单元（MAC）。

不同系列的 DSP 芯片中，各组成部分的具体实现单元数量和具体功能会有所不同。下面介绍其中的算术逻辑单元、累加器、桶形移位寄存器、乘累加单元和寻址单元。

2.3.1 算术逻辑单元（ALU）

DSP 芯片的算术逻辑单元可以实现加/减法运算、逻辑运算等大部分算术和逻辑功能，且许多运算可以在 1 个周期内完成。ALU 的运算结果通常被送往累加器。

ALU 中一般还包括一个进位比特(C)。这个位的值受大多数 ALU 算术操作的影响，它的存在也为支持 ALU 进行更高精度的算术操作提供了可能。C 的值通常可以用指令直接进行设置，也可以利用这个位的值作为判断条件来进行一些操作，如跳转、调用子函数等。

许多 DSP 芯片的 ALU 单元中还包含溢出保护单元。当 ALU 单元的计算结果溢出时，就相应地用正最大值或负最大值来替代计算结果，相关内容请参看第 3 章。这种处理方式在滤波、状态确定等运算中特别有用。

以 TMS320C54x 系列定点 DSP 的 CPU 为例，其中的 ALU 为 40 位，有 2 个输入端和 1 个输出端。当 ALU 进行算术运算时，分为两个 16 位的 ALU 使用，此时来自数据存储器、累加器或 T 寄存器的数据分别进入两个 ALU。在这种情况下，1 个周期内将可以同时完成两个 16 位的操作。TMS320C54x 系列 DSP 的 ALU 功能框图如图 2-2 所示。

图 2-2 中，数据 A 来自累加器 A，数据 B 来自累加器 B，数据 C 来自 CB 数据总线，数据 D 来自 DB 数据总线，数据 M 来自 MAC 单元，数据 S 来自桶形移位器，数据 T 来自 T 寄存器，数据 U 来自 ALU 单元。“/”旁的数字表示总线位宽。以下各图表示方法相同。

浮点芯片中的 ALU 单元与上述的略有不同。为进行单周期的浮点加减法运算，单元中还需集成指数运算逻辑部分。

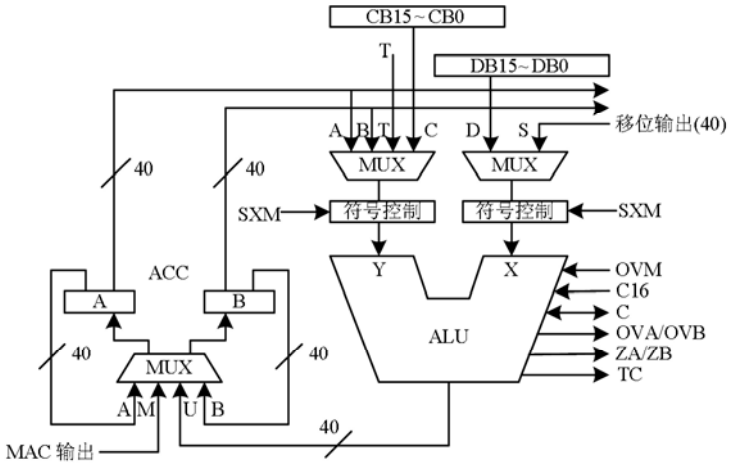


图 2-2　TMS320C54x 系列 DSP 的 ALU 功能框图

2.3.2　累加器

DSP 芯片中需要设置一些特殊的寄存器，用于存放 ALU 单元或其他运算逻辑单元的运算结果，同时作为一些运算逻辑单元的输入，以提供一个中继的功能。一般我们称这种寄存器为累加器。

普通的寄存器与累加器之间存在着本质的区别。一般的寄存器用于存放运算操作的输入值，其位宽度通常与 DSP 的内部数据总线一样；而累加器则用于存放运算操作的结果，其位宽度一般远大于内部总线的位宽度。

TMS320C54x 芯片有 2 个独立的 40 位累加器 ACCA 和 ACCB，可以存放 ALU 或 MAC 单元的运算结果，也可以作为 ALU 的一个输入。每一个累加器分为 3 个部分：保护位（AG 和 BG）、高位字（AH 和 BH）与低位字（AL 和 BL）。其中，保护位可以防止迭代运算中（比如自相关运算）产生的溢出。

TMS320C54x 累加器 ACCA 和 ACCB 结构如图 2-3 和图 2-4 所示。

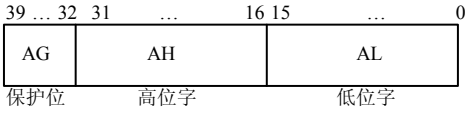


图 2-3　TMS32C54x 累加器 ACCA 结构

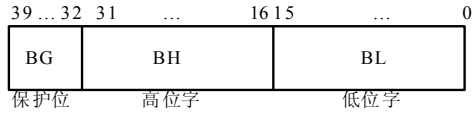


图 2-4　TMS32C54x 累加器 ACCB 结构

AG、BG、AH、BH、AL 和 BL 是存储器映射寄存器（Memory-Mapped Register，MMR），它们的值可以通过压入或弹出堆栈进行保存或恢复。此外，这些寄存器还可以用于寻址操作。ACCA 和 ACCB 的唯一差异在于 ACCA 的（32~16）位可以用作乘累加单元的一个输入。

TMS320C55x 芯片的 CPU 中包含 4 个累加器，其名称为 AC0~AC3。

TMS320C3x 芯片中含有 8 个类似的寄存器，其名称为扩展精度寄存器 R0~R7。在这种寄存器中可以存放 32 位的定点数或 40 位的浮点数，或支持上述两者的运算操作。

TMS320C6000 系列 DSP 中的情况则有一点特殊。在这些芯片中，不是使用超过内部数据总线宽度的累加器来存放逻辑和算术运算的结果，而是采用了寄存器堆的方式，如图 2-5 所示为 TMS320C6000 的寄存器堆。其中，所有的寄存器都能向运算单元提供输入数据，也都能存放算术和逻辑运算结果。当数据是 32 位时，只使用堆中的一个寄存器；当数据是 48 位或 64 位时，使用堆中的一对寄存器来存放数据。注意，图 2-5 中寄存器堆里的每一个小方框表示一对寄存器对。

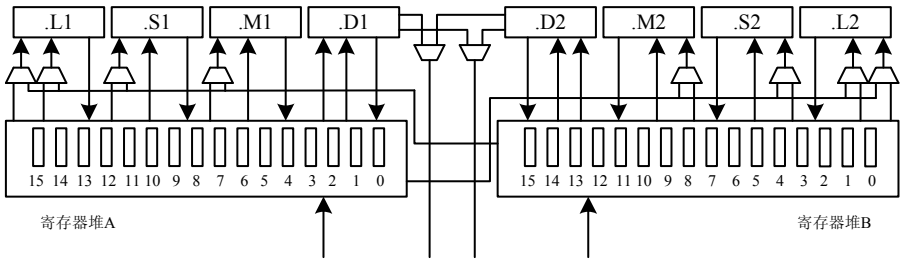


图 2-5　TMS320C6000 的寄存器堆

2.3.3　桶形移位寄存器

桶形移位寄存器主要用于累加器或数据区操作数的移位。不同 DSP 芯片的移位位数范围不同，如 TMS320C54x 系列 DSP 芯片的移位位数为−16~31，其中，移位位数为正则对应于向左移，移位位数为负则对应于向右移。40 位的输出结果可以送到 ALU 的输入端。

根据不同的汇编指令，移位值可以以立即数形式定义，或者存放于状态寄存器中，或者存放于特殊寄存器内。例如，在 TMS320C54x 中：

```
SFTL    A,  +2          ; 累加器 A 中的值逻辑左移 2 位
ADD     A,  ASM,  B     ; 累加器 A 中的值移位（位数由 ASM 值确定）后与累加器 B
                        ; 的值相加，结果放在累加器 B 中
NORM    A               ; 归一化累加器 A 中的值（移位位数由 T 寄存器确定）
```

图 2-6 为 TMS320C54x 的移位寄存器功能框图。

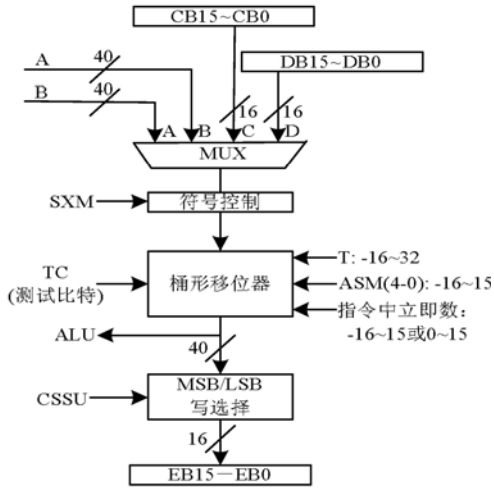


图 2-6　TMS320C54x 的移位寄存器功能框图

2.3.4　乘累加单元

乘累加（MAC）单元包括 1 个乘法器和 1 个专用加法器。MAC 单元具有强大的乘累加功能，在一个流水线周期内可以完成 1 次乘法运算和 1 次加法运算。在滤波以及自相关等运算中，充分利用乘累加指令可以显著提高运算速度。各系列 DSP 芯片的乘法器宽度和专用累加器的宽度不同。如 TMS320C54x 和 TMS320C55x 中的乘法器为 17×17 位，专用累加器为 40 位。

图 2-7 为 TMS320C54x 的乘累加单元功能框图。

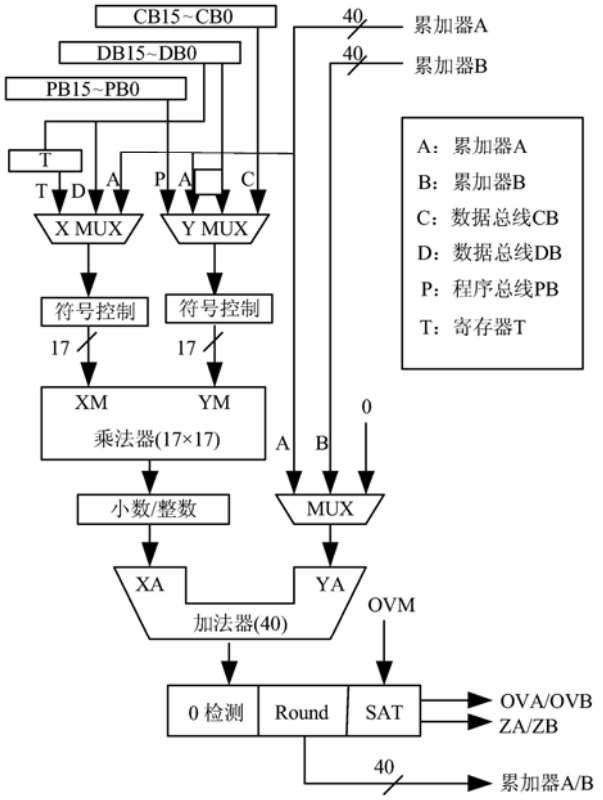


图 2-7　TMS320C54x 的乘累加单元功能框图

在 MAC 单元中，乘法器能够进行有符号数、无符号数以及有符号数与无符号数的乘运算。如，TMS320C54x 系列 DSP 芯片中，依据不同情况作以下三种处理：

- 如果是两个有符号数相乘，则 16 位的乘数与被乘数先进行符号扩展，在最高位前添加 1 个符号位（其值由最高位决定），扩展为 17 位有符号数后再相乘；
- 如果是无符号数乘以无符号数，则在两个操作数的最高位前面添加“0”，扩展为 17 位的操作数再相乘；
- 如果是有符号数与无符号数相乘，有符号数在最高位前添加 1 个符号位（其值由最高位决定），无符号数在最高位前面添加“0”，然后两个操作数相乘。

由于两个 16 位的二进制补码相乘会产生两个符号位，为了提高计算精度，在状态寄存器 ST1 中设置小数模式位 FRCT 为 1，乘法器结果就会自动左移 1 位而去掉 1 个多余的符号位。

在 MAC 单元中，加法器的输入一个来自乘法器的输出，另一个来自累加器 A 或 B 中的某一个输出。加法器的运算结果输出到累加器 A 或 B 中。此外，加法器还包含了舍入和饱和逻辑。舍入将目标累加器的值加上 2^{15}，然后将累加器的低 16 位清零。在汇编指令中，常常在乘累加（MAC）、乘累减（MAS）或乘法（MPY）等指令后加上后缀“R”，表示该指令要将累加器的低 16 位舍入。

TMS320C55x 系列 DSP 芯片中，共有两个 MAC 单元包含在 D 单元中。

需要指出的是，TMS320C6000 系列 DSP 芯片中，并没有乘累加单元，而是在 M 单元中

提供了 16×16 位的乘法和移位功能。

2.3.5 寻址单元

DSP 芯片的程序和数据都是存储在存储器中的，为了迅速有效地访问这些存储器单元，必然涉及寻址问题。各种 DSP 芯片中都提供了多种寻址方式，以提高运算和指令执行效率。

按寻址对象区分，寻址方式主要包括程序寻址和数据寻址两大类。

程序寻址过程中的一个关键单元是程序计数器 PC。PC 中包含需要取指的下一条指令所在的程序存储器地址，一般在存储器地址中按顺序产生。当执行到子函数调用、中断时，将用相应的子函数入口地址或中断服务程序入口地址来加载 PC；当执行到跳转语句时，会使用相应的跳转地址来加载 PC。

PC 一般由程序地址产生单元产生。如 TMS320C54x 系列芯片中，PAGEN（Program-Address Generation logic）用于产生将要取指的指令或存放于程序空间的立即数、系数表等的入口地址。PAGEN 中包含 5 个寄存器，包括 PC、RC、BRC、RSA、REA，所产生的地址通过程序地址总线（Program Address Bus，PAB）与程序存储区相联系。

TMS320C55x 系列 DSP 芯片中，有专门的 P 单元用于程序地址的产生。

TMS320C6000 系列 DSP 芯片中，程序地址的产生由 PMC（Program Memory Controller）单元和 CPU 中的其他相关单元共同完成。

对于数据寻址，指令执行中所涉及的操作数地址一般由指令提供。为了加速数据寻址的速度，减少指令中为操作数地址所保留的比特位数，提高指令执行效率，一般 DSP 芯片都提供了多种的数据寻址方式。

例如 TMS320C54x 系列 DSP 芯片，提供了 7 种高效灵活的数据寻址方式：

- 立即数寻址。操作数为立即数，直接包含在指令中。
- 绝对地址寻址。指令中为固定地址。
- 累加器寻址。将累加器的内容作为地址去访问程序存储器的某个单元。
- 直接寻址。根据所在的存储器的页（DP 或 SP）和页内偏移量（7 比特），确定在数据存储器中的实际地址。
- 间接寻址。通过 8 个辅助寄存器（AR0~AR7）中的地址访问存储器。
- 存储器映射寄存器寻址。这种方式允许直接修改存储器映射寄存器中的值，而不影响当前的 DP 或 SP 的值。
- 堆栈寻址。负责数据在系统堆栈中的压入和弹出。

在执行直接寻址、间接寻址或存储器映射寄存器寻址过程中，通过数据地址产生逻辑（DAGEN）来计算数据存储器中操作数的地址。

TMS320C55x 系列 DSP 芯片中，A 单元专门用于产生数据地址。TMS320C6000 系列 DSP 芯片中，数据地址的产生由 DMC（Data Memory Controller）单元和 CPU 中的其他相关单元共同完成。

寻址方式的更多内容可参考本书的第 7 章。

2.4 总线结构和流水线

大多数的 CPU 都采用冯·诺伊曼结构设计，如图 2-8 所示。在这种结构中，程序指令与

数据共享同一个存储空间，统一编址、依靠指令计数器提供的地址来区分是指令还是数据，采用同一条地址和数据总线进行访问。这种设计便于硬件实现和管理，但每次在读取指令后，还需通过同一条总线来取数，对于需要处理大量数据的 DSP 芯片来说，其数据传输效率不高。因此，大多数 DSP 芯片采用哈佛结构，如图 2-9 所示。

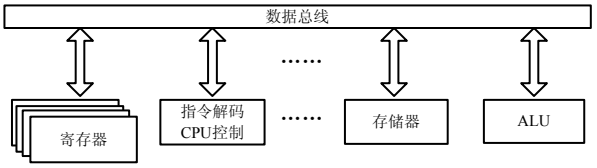


图 2-8　冯・诺伊曼结构

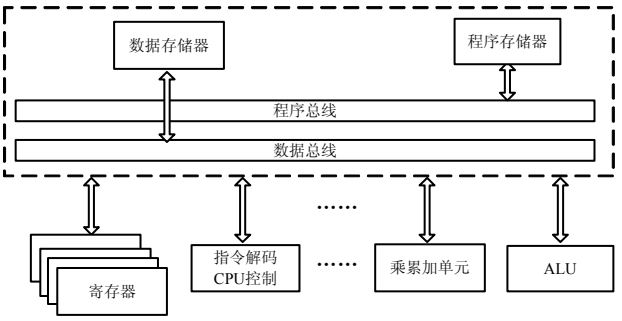


图 2-9　哈佛结构

哈佛结构是不同于传统的冯・诺伊曼结构的并行体系结构，其主要特点是将程序和数据存储在不同的存储空间中，即程序存储器和数据存储器是两个相互独立的存储器，每个存储器独立编址，独立访问。与两个存储器相对应的是系统中设置了程序总线和数据总线两条总线，从而使数据的吞吐率提高了一倍。

在哈佛结构中，由于程序和数据存储器在两个分开的空间中，因此取指和执行能完全重叠运行。为了进一步提高运行速度和灵活性，TMS320 DSP 芯片在基本哈佛结构的基础上作了改进，一是允许数据存放在程序存储器中，并被算术运算指令直接使用，增强了芯片的灵活性; 二是指令存储在高速缓冲器（Cache）中，当执行此指令时，不需要再从存储器中读取指令，节约了一个指令周期的时间。如 TMS320C30 具有 64 个字的 Cache。

与哈佛结构相关，DSP 芯片广泛采用流水线以减少指令执行时间，从而增强了处理器的处理能力。流水线处理的原理是：将指令分为几个子操作，不同子操作由不同的单元完成。这样，每隔一个时钟周期，每个单元就可以进入一条新指令。因此在同一个时钟周期内，在不同的单元可以处理多条指令，相当于并行执行了多条指令。

不同系列的 DSP 芯片的流水线深度是不同的。第一代 TMS320 处理器采用 2 级流水线，第二代采用 3 级流水线，第三代采用 4 级流水线，TMS320C54x 采用 6 级流水线，而 TMS320C6000 系列芯片的流水线深度更深。采用流水线操作，处理器可以并行处理多条指令，每条指令处于流水线上的不同阶段。图 2-10 给出了一个三级流水线操作的例子。

在三级流水线操作中，取指、译码和执行操作可以独立地处理，这可使指令执行能完全重叠。在每个指令周期内，三个不同的指令处于激活状态，每个指令处于不同的阶段。例如，在第 N 个指令取指时，前面一个即第 N–1 个指令正在译码，而第 N–2 个指令则正在执行。一般来说，流水线对用户是透明的。

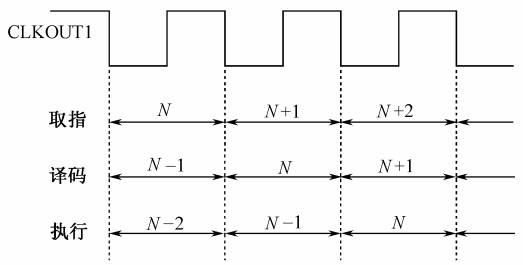


图 2-10　三级流水线操作

2.5　片内存储器

DSP 芯片内部一般都包含片内存储器，用来存储程序代码以及程序运行所需的各种变量。为适应海量的处理数据（如图像处理等场合）对数据空间的需求和复杂的处理算法对程序空间的需求，DSP 芯片也可以在芯片外部扩展存储空间，一般这部分被称为外部存储器。本节简要介绍片内存储器的情况，更多详细内容可参见第 6 章内容。

按照类型，片内存储器包括 ROM、RAM、Flash 等多种类型，其中 RAM 又可以分为两种类型：SARAM（单访问 RAM）和 DARAM（双访问 RAM）；按照使用方式，片内存储器包括程序存储器、数据存储器和 I/O 空间存储器。下面按照类型分别介绍各种片内存储器。

1．片内ROM

片内 ROM 的容量随 DSP 芯片类型的不同而有所变化。它是程序空间的一部分，在一些情况下也可以作为数据空间的一部分来使用。当 DSP 芯片中的 ROM 容量较小时，每次 DSP 芯片复位，ROM 里面包含的引导装载（Bootloader）程序可以把用户程序代码从外部 Flash 或者 EPROM 装入片内 RAM 或外部 RAM 中，从而用户程序能在 RAM 中高速运行。

2．片内Flash

DSP 芯片内部的片内 Flash 容量随芯片类型不同而不同。TI 公司的 DSP 芯片中，在 TMS320(L)F2xxx 系列芯片中集成了 Flash 存储器。

由于片内 Flash 是用来提供永久程序或数据存储空间，并且又可以方便地多次编程，因此受到广泛的欢迎。有的芯片的片内 Flash 具有加密位，如 TMS320LF/LC240xA 系列的 DSP 芯片，这对于保护知识产权非常有用。

3．DARAM

DARAM 由存储器内的一些分块组成。由于每个 DARAM 块在单机器周期内能被访问两次，因此在同一个周期内，CPU 和外设（如 BSP 和 HPI）可以对 DARAM 同时进行读和写操作。DARAM 通常被映射到数据区且用来存储数据，当然也可以映射到程序区用来存储程序代码。

4．SARAM

SARAM 与 DARAM 的区别在于：每个 SARAM 块在 1 个机器周期内只能被访问一次。也就是说，每个机器周期只能进行一次读或者写操作。SARAM 通常被映射到数据区用来存储数据，它也可以映射到程序区用来存储程序代码。

5．存储器映射寄存器

寄存器的使用对于高速算法的好处很明显，基于寄存器，DSP 内核可以直接访问其中的数据，而不必执行多步的存取操作。DSP 芯片中的寄存器包括 CPU 寄存器和外设寄存器，一般都映射到片内的存储器中。

TMS320C2xx 系列 DSP 芯片中的 CPU 寄存器和外设寄存器一般被映射到数据存储器的不同块中，部分寄存器（如 Flash 寄存器）被映射到 I/O 空间中。具体设置可参考不同芯片的手册。

TMS320C54x 系列 DSP 芯片中，数据空间的前 80h 个地址单元内包含有 CPU 和集成外设的映射寄存器。通过访问映射寄存器就可以实现对 CPU 和外设存储器的相应操作。如在调用或中断时，可以完成寄存器内容的保存以及返回时寄存器内容的恢复；还可以完成累加器和其他寄存器之间的数据传送。使用映射寄存器简化了 CPU 和集成外设存储器的访问方式。此外，在数据存储器的 60~7Fh 单元，还有一个 32 字的 DARAM 块。

TMS320C55x 系列 DSP 芯片中，数据空间的前 96 个地址单元内（0000h~005Fh）包含有 CPU 的映射寄存器。在程序空间的前 192 个地址单元内（0000h~00BFh）也有相对应的一个块，建议不要将代码写入这些地址中。

TMS320C55x 系列 DSP 芯片的部分外设寄存器被映射到 I/O 空间中。其中包括 BOOT 模式寄存器、时钟模式寄存器、DMA 控制寄存器、EMIF 寄存器、GPIO 寄存器、Idle 寄存器、指令 Cache 寄存器、McBSP 寄存器、定时寄存器等。

TMS320C6000 系列 DSP 芯片中，寄存器一般都映射在较高地址的片内 RAM 中。

6．Cache

对于 DSP 应用来说，最好的实现方案就是所有程序代码和处理数据都放在片内存储器中，且片内存储器的存取速度与 CPU 相同，以达到最高的执行效率。但是，与 CPU 处理速度的不断提高相比，高速存储器的发展相对较慢，价格很高；同时当前许多应用对 DSP 空间的需求超过了片内存储器空间的增长，这就需要使用存取速度可能相对更慢的片外存储器。这些因素的存在将导致 CPU 的处理迟延，使得 DSP 芯片的处理效率无法得到全部发挥。

为克服这些问题，在一些高速 DSP 芯片中采用了集成 Cache 技术，如图 2-11 所示。

在其中靠近 CPU 的地方放置一个较小但快速的存储器，以避免 CPU 的迟延；在相对较远的地方放置较大但速度也相对较慢的存储器。速度较快的片内存储器就称为 Cache，它由 Cache 控制器管理。通过这种层次化的方式，能够极大地节省数据存取时间，提高 DSP 芯片的执行效率。

目前，部分 TMS320C62x 系列、大部分 TMS320C67x 系列和所有 TMS320C64x 系列 DSP 芯片中都集成了两级 Cache。在 TMS320C55x 系列的部分芯片中也集成了一级 Cache。

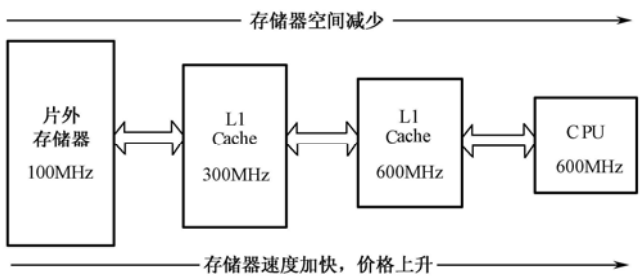


图 2-11　集成 Cache 技术

2.6 集成外设

不同的应用场合，对 DSP 芯片内部集成外设的需求也有所不同。DSP 芯片的集成外设随芯片类型不同而有所变化。通常芯片中可能含有的集成外设有：

- 通用 I/O 接口；
- 定时器；
- HPI 接口；
- 串行口；
- 软件可编程等待状态产生器。

除此之外，DSP 芯片可能还包含 ADC、CAN、USB 等多种类型的外设。在本书第 9 章将对常用外设的使用进行较详细的介绍，不同 DSP 芯片的外设配置及使用的详尽情况请参考相应的芯片手册。

2.6.1 通用 I/O 接口

大多数 DSP 芯片中都设置了通用 I/O 口（GPIO）。GPIO 与定时、中断和各种时序的有机结合，可以模拟出绝大多数的串口通信方式。

TMS320C2xx、TMS320C54x 和 TMS320C55x 系列的大部分芯片中，都有一个 XF 引脚作为外部标志输出引脚，XF 的电平值受状态寄存器 ST1 中的 XF 位控制。

TMS320C54x 系列芯片的$\overline{\text{BIO}}$引脚是转移控制输入引脚，用于监视外部器件的状态。在汇编指令中，有判断$\overline{\text{BIO}}$引脚状态并产生相应条件转移的指令。

除了 XF 和$\overline{\text{BIO}}$可以作为通用 I/O 引脚以外，大部分 DSP 芯片还将某些数字输入/输出引脚作为通用 I/O 引脚。这些引脚的输入/输出状态可以通过状态寄存器来设置。

2.6.2 定时器

片内定时器是一个软件可编程的计数器，通常包括以下 3 个基本的寄存器：定时周期寄存器、定时控制寄存器和定时寄存器。通过设定定时周期寄存器、定时控制寄存器中的值可以改变定时间隔，详见第 9 章。

2.6.3 主机接口（HPI）

为了便于与标准的微处理器总线连接，TI 公司的 TMS320C54x、TMS320C55x 和 TMS320C6000 系列 DSP 芯片提供了主机接口 HPI（Host Port Interface）。HPI 接口通过 HPI 控制寄存器（HPIC）、地址寄存器（HPIA）、数据锁存器（HPID）和 HPI 内存块实现与主机通信。其主要特点有：接口所需外围硬件很少；HPI 单元允许芯片直接利用一个或两个数据选通信号、一个独立或复用的地址总线、一个独立或复用的数据总线接到微控制单元 MCU 上；主机和 DSP 可独立地对 HPI 接口操作；主机和 DSP 握手可通过中断方式来完成。另外，主机还可以通过 HPI 接口装载 DSP 应用程序、接收 DSP 运行结果或诊断 DSP 运行状态。HPI 为 DSP 芯片的接口开发提供了一种极为方便的途径。

根据数据总线位数的不同，HPI 可以按照 HPI-8 和 HPI-16 两种方式工作。

2.6.4　串行口

为了将 DSP 芯片与外围器件（如 A/D 转换器、编解码器等）通过串行口连接起来，大多数 DSP 芯片中都包含同步的串行通信接口。

同步串行口的基本信号包括以下四种：数据时钟 CLK、帧同步 FS、数据接收 DR 和数据发送 DX。数据时钟信号用作串行数据位的移入或移出参考。帧同步用于指示 8 位、16 位、32 位数据字的开始。数据接收和数据发送用于传送数据信号。

为适应内、外部接收时钟的不同，通常时钟信号会分为输入时钟 CLKR 和输出时钟 CLKX，两个时钟可以分别由外部器件提供，也可以分别由 DSP 芯片产生。同样，帧同步信号也可以分为接收帧同步 FSR 和发送帧同步 FSX，这两个信号也可以由外部提供或由 DSP 芯片自主产生。此时，同步串行接口扩展为 6 个信号。图 2-12 给出了串行口信号的连接关系，其中的时钟信号和帧同步信号都由 DSP 芯片提供，图 2-13 给出了串行口信号的一种时序关系。

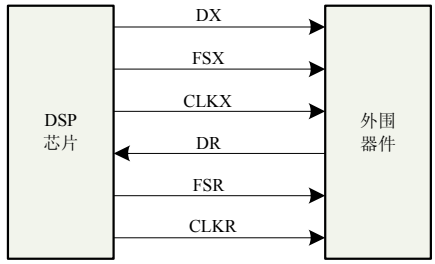


图 2-12　串行口信号的连接关系

同步串行口的接收和发送帧的比特位数、发送与接收时钟频率等可以通过控制寄存器来配置。串行口在接收或发送一个完整的帧后，通过内部中断来通知 DSP 内核进行处理。一些 DSP 芯片内部集成了 DMA 处理器，可以在用户指定的存储器位置一次传送多个帧数据，这种方式可以有效降低 DSP 内核的处理开销。

缓冲串行口（BSP）是一种增强型的标准串行口，它由一个全双工双缓冲串行口和一个自动缓冲单元（ABU）组成。由于其中的串行口与标准串行口的功能相同，因此在标准模式下，缓冲串行口的操作与标准串行口的工作方式是一样的。在自动缓冲模式下，BSP 使用 ABU 内嵌式的地址产生器，串行口和缓冲区之间可以直接进行数据传输，缓冲区的起始地址和容量是软件编程控制的。ABU 可以使得在没有 CPU 控制的情况下直接对缓冲区进行读写，从而串行口收发数据的开销降至最低，能以 CLKOUT 的速率全速传输数据。

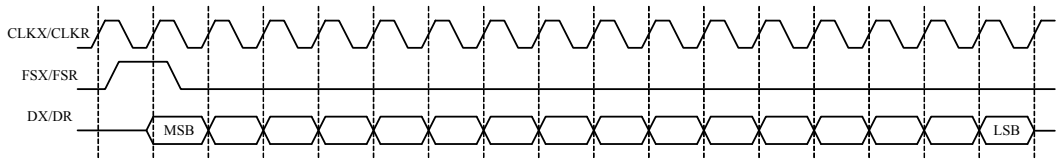


图 2-13　串行口信号的时序关系

TDM 串行口采用时分复用方式将多个外部器件复用与 DSP 芯片进行串行通信，每一个时隙对应于其中的一路通信。这样 TDM 就可以和外部的多个应用接口实现方便灵活的数据交换。

多通道缓冲串行口（McBSP）是在缓冲串行口的基础上发展起来的，其功能包括：

- 全双工通信；
- 双缓冲数据寄存器，允许连续的数据流；
- 收发独立的帧信号和时钟信号；
- 可与工业标准的编/解码器、AICs（模拟接口芯片）以及其他串行 A/D、D/A 接口；
- 数据传输可以利用外部时钟，或者是片内的可编程时钟；
- 当利用 DMA 为 McBSP 服务时，串行口数据读写具有自动缓冲能力。

2.6.5 软件可编程等待状态产生器

为克服 DSP 芯片内部总线的高速与外围设备数据传输速率低之间的矛盾，大多数 DSP 芯片内部都集成了软件可编程等待状态产生器，通过设置相应的寄存器值，可将外部总线周期扩展到多个机器周期，以方便 DSP 芯片的内核与慢速的片外存储器和 I/O 器件接口。一些 DSP 芯片中还可以针对同一类存储器的不同块设置相应的等待周期。

2.7 TI 定点 DSP 芯片

自 1982 年 TI 推出第一片定点 DSP 芯片 TMS32010 以来，TI 的定点 DSP 芯片已经经历了 TMS320C1x、TMS320C2x、TMS320C5x 、TMS320C2xx、TMS320C54x、TMS320C62xx/C64xx 等几代产品，产品的性能价格比不断提高，应用越来越广泛。目前应用广泛的主要有 TMS320C2000 系列、TMS320C5000 系列和 TMS320C6000 系列中的 TMS320C62xx/C64xx。下面我们分别来介绍这些芯片的主要特征。

2.7.1 早期的定点 DSP 芯片

2.7.1.1 TMS320C1x

1．基本特征

第一代 TMS320 系列 DSP 芯片包括：TMS32010、TMS32011、TMS320C10、TMS320C15/E15 和 TMS320C17/E17。其中，TMS32010 和 TMS32011 采用 2.4 μm 的 NMOS 工艺，而其他几种则采用 1.8 μm CMOS 工艺。这些芯片的典型工作频率为 20 MHz。TMS320 第一代 DSP 芯片的主要特征如下：

- 指令周期：160 ns/200 ns/280 ns；
- 片内 RAM：144 字/256 字（TMS320C15/E15/C17/E17）；
- 片内 ROM：1.5 K/4 K 字（TMS320C15/C17）；
- 4 K 字的片内程序 EPROM（TMS320E15/E17）；
- 4 K 字外部全速存储器扩展；
- 并行乘法器：乘积为 32 位；
- 桶形移位器：将数据从存储器移到 ALU；
- 并行移位器；
- 允许文本交换的 4×12 位堆栈；

- 两个间接寻址的辅助寄存器；
- 双通道串行口（TMS32011,TMS320C17/E17）；
- 片内压扩硬件（TMS32011,TMS320C17/E17）；
- 协处理器接口（TMS320C17/E17）；
- 器件封装：40 脚双列直插（DIP）/44 脚塑封（PLCC）。

2．TMS320C10

TMS320 DSP 芯片的第一代产品基于 TMS32010 和它的 CMOS 版本 TMS320C10 的结构。TMS32010 于 1982 年推出，是第一个能够达到 5 个 MIPS 的微处理器。

TMS32010 采用改进的哈佛结构，即程序和数据空间相互独立。程序存储器可在片内（1.5 K 字）或片外（4 K 字）。片内数据 RAM 为 144 字。有四个基本的算术单元：算术逻辑单元（ALU）、累加器（ACC）、乘法器和移位器。

（1）ALU：32 位数据操作的通用算术逻辑单元。ALU 可进行加、减和逻辑运算。

（2）ACC：累加器存储 ALU 的输出，也是 ALU 的一个输入。它采用 32 位字长操作，分高 16 位和低 16 位两部分。处理器提供高 16 位和低 16 位的专门指令：SACH（高 16 位）和 SACL（低 16 位）。

（3）乘法器：16×16 位并行乘法器由三个单元组成：T 寄存器、P 寄存器和乘法器阵列。T 寄存器存储被乘数，P 寄存器存储 32 位乘积。为了使用乘法器，被乘数首先必须从数据 RAM 中装入到 T 寄存器，可用 LT、LTA 和 LTD 指令。然后执行 MPY（乘）或 MPYK（乘立即数）指令。乘和累加器操作可用 LTA、LTD 和 MPY、MPYK 指令在两个指令周期内完成。

（4）移位器：有两个移位器可用于数据移位。一个是桶形移位器，另一个是并行移位器。桶形移位器又称定标移位器。当数据存储器的数据送入累加器或与累加器中的数据进行运算时，先通过这个移位器进行 0~6 位左移，然后再进行运算。并行移位器，即输出移位器，其作用是将累加器中的数据左移 0、1 或 4 位后再送入数据存储器中以实现小数运算或小数乘积的调整。

在 TMS32010/C10 基础上，又派生了多个处理器，它们可提供不同的处理器速度、存储器扩展和各种 I/O 集成。

3．TMS320C1x的其他芯片

（1）TMS320C15/E15

TMS320C15/E15 与 TMS32010 的代码和引脚全兼容，提供 256 个字的片内 RAM 和 4 K 字的片内 ROM（C15）或 EPROM（E15）。TMS320C15 有 200 ns 和 160 ns（TMS320C15-25）两种。

（2）TMS320C17/E17

TMS320C17/E17 是一个专用的微处理器。它有 4 K 字的片内程序 ROM（C17）或 EPROM（E17），一个全双工串行通信的双通道串行接口，片内硬件压扩器（μ律/A 律），一个用于串行通信的串行接口定时器，及一个协处理器接口。协处理器接口可以在处理器和 4/8/16 位微处理器之间提供直接接口。TMS320C17/E17 与 TMS32010 目标代码兼容，且可用相同的开发工具。该处理器是基于 TMS320C10 的 CPU 内核，外加片内的外设存储器和 I/O 模块。TMS320C17/E17 可认为是半定制的 DSP 芯片。表 2-1 是 TMS320 第一代产品的比较表。

表 2-1　TMS320 第一代处理器

DSP 芯片	指令周期/ns	制造工艺	片内程序 ROM/字	片内程序 EPROM/字	片内数据RAM/字	片外程序/字
TMS32010	200	NMOS	1.5K		144	4K
TMS32010-25	160	NMOS	1.5 K		144	4 K
TMS32010-14	280	NMOS	1.5 K		144	4 K
TMS32011	200	NMOS	1.5 K		144	
TMS320C10	200	CMOS	1.5 K		144	4 K
TMS320C10-25	160	CMOS	1.5 K		144	4 K
TMS320C15	200	CMOS	4 K		256	4 K
TMS320C15-25	160	CMOS	4 K		256	4 K
TMS320E15	200	CMOS		4 K	256	4 K
TMS320C17	200	CMOS	4 K		256	
TMS320C17-25	160	CMOS	4 K		256	
TMS320E17	200	CMOS		4 K	256	

2.7.1.2　TMS320C2x

第二代 TMS320 DSP 芯片包括 TMS32020、TMS320C25/E25、TMS320C26 及 TMS320C28。在这些芯片中，TMS32020 是一个过渡的产品，其指令周期为 200 ns，与 TMS32010 相当，而其硬件结构则与 TMS320C25 一致。在第二代 DSP 芯片中，TMS320C25 是一个典型的代表，其他芯片都是由 TMS320C25 派生出来。其中 TMS320E25 将内部 4K 字的 ROM 改为 EPROM，TMS320C26 去掉了内部的 4K ROM，而将片内 RAM 增加到 1.5 K 字。TMS320C28 则将内部 ROM 增加到 8 K 字。由于 TMS320C25 的典型性，下面讨论 TMS320C25 的基本特征和结构。

1．TMS320C25 的基本特征

- 指令周期：TMS320C25：100 ns，TMS320C25-33：120 ns，TMS320C25-50：80 ns；
- 片内掩膜 ROM：4 K 字；
- 片内 RAM：544 字，分 B0、B1 和 B2 三块；
- 程序和数据空间均为 64 K 字；
- 具有 8 级硬件堆栈、8 个辅助寄存器；
- 具有全静态双缓冲串行口，可与许多串行器件直接接口；
- 与低速片外存储器通信的等待状态插入；
- 采用 HOLD 操作的 DMA；
- FFT 变换用的比特反转寻址；
- 扩展精度算术和自适应滤波支持；
- 从外部存储器全速执行的 MAC/MACD 指令；
- 具有在多处理器之间进行同步的能力，支持多处理器共享存储器；
- 1.8 μm CMOS 工艺，68 脚 PGA 或 PLCC 封装。

2．TMS320C25 的结构

TMS320C25 是第二代 TMS320DSP 芯片中与 TMS32020 引脚全兼容的 CMOS 版本，但指令执行速度是 TMS32020 的 2 倍，且增加了硬件和软件资源。指令集是 TMS32010 和 TMS32020 的超集，在源代码级与它们兼容。此外，在代码级与 TMS32020 兼容，因此，TMS32020 的程

序可不用修改直接在 TMS320C25 上运行。

100 ns 的指令周期可提供较强的运算能力。由于大部分指令在一个指令周期内执行，处理器在一秒内可执行一千万条指令（10 个 MIPS）。运算能力的增加主要得益于乘/累加指令（MAC）和带数据移位的乘/累加指令（MACD）、具有专用算术单元的 8 个辅助寄存器、适合于自适应滤波和扩展精度运算的指令集支持、比特反转寻址、快速的 I/O 支持等。

指令集中提供在两个存储空间中进行数据移动的指令。在外部，程序和数据存储空间在同一总线上复用，从而使得在尽量减少芯片引脚的情况下最大程度地扩大两个空间的地址范围。在内部，TMS320C25 结构是通过保持程序和数据总线结构分离以使指令全速执行来获得最大的处理能力。

芯片内部的程序执行采用三级流水线形式。流水线对用户来说是透明的。但是，在某些情况下流水线可能被打断，如跳转指令，在这种情况下，指令执行时间要考虑流水线必须清空和重新填充。两块大的片内 RAM 在系统设计时提供了很大的灵活性，其中一块既可配置为程序也可配置为数据存储器。片外 64 K 字的数据空间可直接寻址，从而使 DSP 算法实现更为方便。片内 4 K 字的掩膜 ROM 可用来降低系统成本，若程序不超过 4K 字，可使 TMS320C25 不用扩展片外程序存储器。剩下的 64 K 字程序空间在片外，大量的程序可在片外存储器中全速运行。程序也可从片外慢速 EPROM 中装入到片内高速 RAM 中全速运行。此外，还有硬件定时器、串行口和数据块搬移功能。

下面介绍 TMS320C25 的存储器分配、中央算术逻辑单元（CALU）、硬件乘法器、控制操作、串行口和 I/O 接口。

（1）存储器分配：TMS320C25 具有 4 K 字的片内程序 ROM 和 544 字的片内 RAM。RAM 分为三块：B0、B1、B2。其中，B0 块（256 字）既可配置为数据存储器（用 CNFD 指令），也可配置为程序存储器（用 CNFP 指令）。其余 288 字（B1 和 B2 块）只能是数据存储器。544 字的片内 RAM 可使 C25 能处理 512 字的数据阵列，如可进行 256 点复数 FFT 运算，且尚有 32 字用作中间结果的暂存。TMS320C25 提供片外可直接寻址的程序和数据空间各 64 K 字。

寄存器组包含 8 个辅助寄存器（AR0~AR7），它们可用作数据存储器的间接寻址和暂存，从而可增加芯片的灵活性和效率。这些寄存器既可用指令直接寻址，也可用 3 比特的辅助寄存器指针（ARP）间接寻址。辅助寄存器和 ARP 既可从数据存储器装数，也可装入立即数。寄存器的内容也可存入数据存储器中。辅助寄存器组与辅助寄存器算术单元（ARAU）相连接，用 ARAU 访问信息表无需 CALU 参与地址操作，这样可让 CALU 进行其他操作。

（2）中央算术逻辑单元：CALU 包含有一个 16 位的定标移位器（Scaling），一个 16×16 位的并行乘法器，一个 32 位的累加器和一个 32 位的算术逻辑单元（ALU）。移位器根据指令要求提供 0 到 16 位的数据左移。累加器和乘法器输出端的移位器适合于数值的归一化、比特提取、扩展精度算术和溢出保护。

典型的 ALU 指令实现包含以下三步：

- 数据在数据总线上从 RAM 中获取；
- 数据移交给完成算术运算的定标移位器和 ALU；
- 结果送回累加器。

32 位累加器可分为二个 16 位以进行数据存储：SACH（高 16 位）和 ACCL（低 16 位）。累加器有一个进位位可方便加法和减法的多精度运算。

（3）硬件乘法器：TMS320C25 具有一个 16×16 位的硬件乘法器，它能在一个指令周期内计算一个 32 位乘积。有两个寄存器与乘法器相关：①16 位暂存寄存器 TR，用于保存乘法器的一个操作数；②32 位乘积寄存器 PR，用于保存乘积。

乘积寄存器的输出可左移 1 位或 4 位，这对于实现小数算术运算或调整小数乘积很有用。PR 的输出也可右移 6 位，这样可连续执行 128 次乘/加而无溢出。无符号乘（MPYU）指令方便扩展精度乘法。

（4）I/O 接口：I/O 空间由 16 个输入口和 16 个输出口组成。这些口可提供全 16 位并行 I/O 接口。输入（IN）和输出（OUT）操作典型的是 2 个周期，但若用重复指令，可变成单周期指令。I/O 器件映射到 I/O 地址空间，其方式与存储器映射方式相同。与不同速度的存储器或 I/O 器件接口采用 READY 线完成。

TMS320C25 也支持外部程序/数据存储器的 DMA，其他处理器通过置$\overline{\text{HOLD}}$为低后可完全控制 TMS320C25 的外部存储器，使其将地址、数据和控制线呈高阻状态。外部处理器和 TMS320C25 之间的通信可通过中断来完成。TMS320C25 芯片提供两种 DMA 方式，一种是加上 HOLD 后停止执行；另一种是继续执行，但执行是在片内 ROM 和 RAM 中进行，这可大大提高其性能。

3．TMS320C25 的指令系统

TMS320C25 的指令总共有 133 条，其中的 97 条是单周期指令。在另外的 36 条指令中，21 条是跳转、调用、返回等指令，这些指令需重新装入程序计数器，使执行流水线中断。另外 7 条指令是双字和长立即数指令。剩下的 8 条指令支持 I/O 操作、存储器之间的数据交换或提供处理器内部额外的并行操作。而且，这 8 条指令（IN，OUT，BLKD，BLKP，TBLR，TBLW，MAC，MACD）与重复计数器配合使用时可成为单周期指令。这主要利用了处理器的并行机制，使得复杂的计算可用很少的几条指令来完成。

由于大多数指令用单 16 位字编码，故可在一个周期内完成。存储器寻址方式有三种，直接寻址、间接寻址和立即寻址。直接和间接寻址都用来访问数据存储器，立即数寻址利用由程序计数器确定的存储器内容。

使用直接寻址方式时，指令字的 7 位和 9 位数据存储器页指针（DP）构成 16 位的数据存储器地址。其中，每页长 128 字，共有 512 页，故可寻址 64 K 的数据空间。间接寻址借助于 8 个辅助寄存器（AR0~AR7）。表 2-2 列出了 7 种间接寻址方式。其中的比特反转寻址可大大提高 FFT 运算的 I/O 效率。其中，OP 表示某种运算，NARP 表示新的 ARP。

表 2-2　TMS320C25 的寻址方式

寻址方式	操作
OP A	直接寻址
OP *（,NARP）	间接寻址；AR 不变
OP *+（,NARP）	间接寻址；当前 AR 加 1
OP *−（,NARP）	间接寻址；当前 AR 减 1
OP *0+（,NARP）	间接寻址；AR0 加到当前 AR
OP *0−（,NARP）	间接寻址；当前 AR 减去 AR0
OP *BR0+（,NARP）	间接寻址；AR0 加到 AR
OP *BR0−（,NARP）	间接寻址；当前 AR 减去 AR0

2.7.1.3　TMS320C5x

TMS320C5x 是 TI 公司的第五代产品，是继 TMS320C1x 和 TMS320C2x 之后的第三代定点 DSP 处理器。它的核心中央处理器（CPU）以 TMS320C25 的核心 CPU 为基础，增强型结构大幅度地提高了整体性能。TMS320C5x 工作速度是 TMS320C25 的两倍以上，对于 TMS320C1x 和 TMS320C2x 具有源代码向上兼容特性。这种兼容性保留了过去开发的软件，便于系统升级到更高性能的 DSP 系统。TMS320C5x 系列有 TMS320C50/C51/C52/C53 等多种产品，它们的主要区别是片内 RAM、ROM 等资源的多少，如 TMS320C50 内部具有 10 K RAM 和 2 K ROM，其中 2 K ROM 已经固化了引导程序。TMS320C51 内部具有 2 K RAM 和 8 K ROM。TMS320C52 内部具有 1 K RAM 和 4 K ROM，减少了一个串行口。TMS320C53 内部则有 4 K RAM 和 16 K ROM。芯片的其他性能则是完全相同的。下面我们以 TMS320C50 为例介绍 TMS320C5x DSP 芯片。

1．TMS320C50 的基本特征

TMS320C50 的主要特点包括：

- 25/35/50 ns 的指令周期（20~40 MIPS）；
- 224 K×16 位最大可寻址外部存储空间（64 K 程序、64 K 数据、64 K I/O、32 K 全局）；
- 算术逻辑单元（ALU），32 位累加器（ACC）以及 32 位加法器的缓冲器（ACCB）；
- 并行逻辑单元（PLU）；
- 结果具有 32 位的 16×16 位并行乘法器；
- 单周期乘/累加指令；
- 具有一个专用算术单元的 8 个辅助寄存器，可用于间接寻址；
- 8 级硬件堆栈；
- 0~16 位数据左移和右移；
- 两个间接寻址的循环缓冲器，用于循环寻址；
- 程序代码的单指令重复和程序块重复；
- 全双工同步串行口，用于完成 TMS320C5x 与其他串行器件之间的直接通信；
- 时分多址访问（TDM）串行口；
- 内部定时器，可用软件控制；
- 64 K 并行 I/O 口，其中 16 个有存储器映像；
- 可软件编程的等待状态发生器；
- 扩展保持操作，用于并发外部 DMA；
- 四级流水线操作，用于延迟跳转、调用、返回指令；
- 比特反转寻址方式，用于 FFT 运算；
- JTAG 扫描仿真接口（IEEE 标准，P1149.1）。

（1）CPU

TMS320C5x CPU 的增强功能在提高性能和通用性的同时，保持了对 TMS320C1x 和 TMS320C2x 源代码的兼容性。硬件的改进包括一个 32 位累加器缓冲器，附加定标能力，利用附加硬件功能的新指令。新的控制功能包括独立的并行逻辑单元（PLU）和一组文本交换寄存器。数据管理方面的改进包括采用新的块搬移指令和存储器映像寄存器指令。TMS320C50 有 28 个存储器映像寄存器和 16 个存储器映像的 I/O 口。

（2）片内 ROM

TMS320C50 拥有 2K×16 位掩蔽 ROM，内部固化了引导程序。引导程序可把程序从外部 ROM/EPROM、串行口或并行 I/O 口引导至运行速度较快的 SRAM 中。这块引导 ROM 可通过 PMST 状态寄存器中的 MP/$\overline{\text{MC}}$位从程序存储空间去除。如果该 ROM 未选，TMS320C50 由片外存储器启动执行。

（3）片内数据 RAM

TMS320C50 具有 1056 字的片内 RAM，这块 RAM 可在每个机器周期内访问两次（双寻址 RAM），只要两次访问不是“写”操作。这块存储空间主要用于存储数据，但是如果需要也可用于存储程序和数据。其配置有两种方式：所有的 1056 字都作为数据存储区，或者将其中的 544 字作为数据存储器，512 字作为程序存储区。可通过状态寄存器 ST1 中的 CNF 位选择设置。

（4）片内程序/数据 RAM

TMS320C50 还具有 9K 字的片内 RAM。这一存储区可以由软件设置映射到程序或数据存储空间。程序从片外存储器引导后，可装入到该存储区全速运行。

（5）片内存储器安全

TMS320C50 可以通过可屏蔽选择来保护片内存储器的内容。当相关比特置位时，外部无法访问片内存储空间。

（6）有地址映射的软件等待状态发生器

软件等待状态逻辑不需要任何外部硬件就可以实现 TMS320C50 与速度较慢的片外存储器和 I/O 设备接口。该电路系统拥有可产生 16 个等待状态的发生器，其中可由用户编程操作的有 0、1、2、3 和 7 个状态。

（7）并行 I/O 口

TMS320C50 共有 64K I/O 口，其中的 16 个可映射在数据存储空间。这些口可由 IN 或 OUT 指令寻址。具有存储器映像的 I/O 口可按存储器的读写方式访问。I/O 口的访问由$\overline{\text{IS}}$线选通。增加简单的片外地址译码电路，就可实现 TMS320C50 的 I/O 口与外部 I/O 设备的简单连接。

（8）串行 I/O 口

TMS320C50 有两个高速串行口。串行口最快可按 1/4 机器周期（CLKOUT1）操作。一个是同步全双工串行口，发送和接收都有缓冲，分别由可屏蔽外部中断信号控制。数据可按照 8 或 16 位方式传输。另一个串行口为全双工串行口，可设置为同步方式也可以设置为时分多址（TDM）方式。TDM 串行口一般用于多处理机系统。

（9）16 位硬件定时器

16 位硬件定时器可由软件进行控制，通过设置相应的状态位，定时器可工作在停止、重启动、复位或不工作等状态。

（10）用户可屏蔽中断

TMS320C50 有四个外部中断线。片内的中断锁存电路可实现异步中断操作。此外，还有 5 个内部中断：一个定时器中断和 4 个串行口中断。

（11）JTAG 扫描逻辑

JTAG 扫描逻辑电路用于仿真和测试，采用 JTAG 可实现在线仿真。

2．TMS320C50 的结构

TMS320C50 采用了与 TMS320C25 类似的哈佛总线结构，通过两个独立的数据总线和程序总线，最大限度地提高运算速度。指令支持在两个存储区之间的数据传输。

TMS320C50 使用 32 位的 ALU 和累加器以实现 2 的补码运算。ALU 是一个通用的算术单元，它所使用的运算数据（16 位）来自数据 RAM 或来自立即数指令，也可以是乘积寄存器中 32 位的乘积结果。除此之外，ALU 还可以执行位操作。累加器保存 ALU 的输出，也为 ALU 提供下一个输入。字长为 32 位的累加器分为高字节和低字节。指令提供对低字节和高字节的单独存储。32 位的累加器缓冲器用于临时快速地存储累加器的内容。

除了 ALU 之外，并行逻辑单元（PLU）可以在不影响累加器内容的情况下，对数据进行逻辑操作。PLU 提供了高速控制器需要的位处理能力，并简化了控制和状态寄存器需要的置位、清零和测试操作。

乘法器以单指令周期完成 16×16 位的乘法，结果为 32 位。乘法器由三部分组成：分别是乘法器阵列、PREG（乘积寄存器）、TREG0（临时寄存器）。16 位的 TREG0 存储乘数，PREG 保存 32 位的乘积结果。乘法器中的数值来自数据存储器，当使用 MAC/MACD/MADS/MADD 指令时来自程序存储器，或者来自乘立即数指令（MPY #）。片内快速乘法器对执行诸如卷积、相关和滤波等基本的 DSP 操作十分有效。

TMS320C50 的定标移位器有一个 16 位的输入来自数据总线，32 位的输出连接到 ALU。定标移位器依照指令的编程使输入数据产生 0 到 16 位的左移。移位量取决于指令或移位计数寄存器（TREG1）中的定义值。输出的最低有效位（LSB）补零，最高有效位补 0 或符号扩展（取决于状态寄存器 ST1 的符号扩展模式位 SXM）。附加的移位能力使处理器能执行数值定标、二进制位提取、符号扩展运算和溢出防止等功能。

8 级硬件堆栈用于在中断及子程序调用时保存程序计数器的内容。中断发生时，重要寄存器（ACC、ACCB、ARCR、INDX、PMST、PREG、ST0、ST1 和 TREG）压入堆栈，中断返回时弹出，实现了无开销的中断文本切换。

2.7.2　TMS320C2000 系列

TMS320C2000 系列包括 TMS320C20x、TMS320C24x 和 TMS320C28x 三类。

TMS320C20x 是继 TMS320C2x 和 TMS320C5x 之后出现的一种低价格、高性能定点 DSP 芯片。TMS320C20x 系列 DSP 芯片具有如下特点：

（1）处理能力强。最高运算能力为 40 MIPS。

（2）片内具有较大的 Flash 存储器。TMS320C20x 是最早使用 Flash 存储器的 DSP 芯片。Flash 存储器具有比 ROM 灵活、比 RAM 便宜的特点。TMS320F206 和 TMS320F207 片内具有 32K 字的 Flash 存储器和 4.5 K 字的 RAM。利用 Flash 存储器存储程序，不仅降低了成本，减小了体积，同时系统升级也比较方便。

（3）功耗低。TMS320C20x 系列 DSP 芯片在 5 V 工作时每个 MIPS 消耗 1.9 mA 电流，在 3.3 V 工作时每个 MIPS 消耗 1.1 mA 电流。使用 DSP 核的省电模式可进一步降低功耗。

（4）资源配置灵活。

表 2-3 是 TMS320C20x 系列 DSP 芯片的资源配置比较表。

表 2-3　TMS320C20x 系列芯片的资源配置比较表

TMS320 C20x	指令周期（ns）	ROM（字）	RAM（字）	Flash（字）	同步串行口	异步串行口
C203	25/35/50	–	544	–	1	1
C204	25/35/50	4 K	544	–	1	1
C205	25/35/50	–	4.5 K	–	1	1
F206	25/35/50	–	4.5 K	32 K	1	1
F207	25/35/50	–	4.5 K	32 K	2	1
C209	35/50	4 K	4.5 K	-	-	-

TMS320C24x 系列 DSP 芯片针对数字控制系统应用作了优化设计，芯片内部具有多达 16 路的 10 位模数转换功能，具有多个通用定时器和一个监视（Watchdog）定时器，具有多达 16 个通道的 PWM（Pulse Width Modulation）通道，最多具有 41 个通用输入输出引脚。表 2-4 列出了 TMS320C24x 系列芯片的资源配置比较表。

表 2-4　TMS320C24x 系列芯片的资源配置比较表

TMS320 C24x	速度 MIPS	RAM（字）	ROM（字）	Flash（字）	I/O 引脚	比较/PWM 通道	定时器	同步串行口	异步串行口	A/D 通道数/转换时间(μs)
F240	20	544	–	1 6 K	28	9/12	3/1	1	1	16ch/6.6
C240	20	544	16 K	–	28	9/12	3/1	1	1	16ch/6.6
F241	20	544	–	8 K	26	5/8	2/1	1	1	8 ch/0.85
C242	20	544	4 K	–	26	5/8	2/1	–	1	8 ch/0.85
F243	20	544	–	8 K	32	5/8	2/1	1	1	8 ch/0.85
LF2407	30/40	2.5 K	–	32 K	41	10/16	4/1	1	1	16 ch/0.5
LF2406	30/40	2.5 K	–	32 K	41	10/16	4/1	1	1	16 ch/0.5
LF2402	30/40	544	–	8 K	21	5/8	2/1	–	1	8 ch/0.5
LC2406	30/40	2.5 K	32 K	–	41	10/16	4/1	1	1	16 ch/0.5
LC2404	30/40	1.5 K	16 K	–	41	10/16	4/1	1	1	16 ch/0.5
LC2402	30/40	544	4 K	–	21	5/8	2/1	–	1	8 ch/0.5

注：表中定时器一栏中，左边的数字表示通用定时器的数量，右边的数字表示 watchdog 定时器的数量。

TMS320C28x 是到目前为止用于数字控制领域性能最好的 DSP 芯片。这种芯片采用 32 位的定点 DSP 核，最高速度可达 400 MIPS，可以在单个周期内完成 32×32 位的乘累加运算，具有增强的电机控制外设、高性能的模数转换能力和改进的通信接口，具有 8 G 字节的线性地址空间，采用低电压供电（3.3 V 外设/1.8 V 核），与 TMS320C24x 源代码兼容。该系列芯片正在陆续开发之中。

TMS320C2000 系列 DSP 芯片价格低，具有较高的性能和适用于控制领域的必要功能，因此可广泛应用于工业自动化、电机控制、家用电器和消费电子等领域。



2.7.3　TMS320C5000 系列

TMS320C500 系列 DSP 芯片目前包括了 TMS320C54x 和 TMS320C55x 两大类。这两类芯片软件完全兼容，所不同的是 TMS320C55x 具有更低的功耗和更高的性能。

TMS320C54x 是为实现低功耗、高性能而专门设计的定点 DSP 芯片，其主要应用是无线通信等应用系统。该芯片的内部结构与 TMS320C5x 不同，因而指令系统与 TMS320C5x 和 TMS320C2x 是互不兼容的。

TMS320C54x 的主要特点包括：

（1）运算速度快。运算速度为 30~532 MIPS。

（2）优化的 CPU 结构。内部有一个 40 位的算术逻辑单元，两个 40 位的累加器，两个 40 位加法器，一个 17×17 的乘法器和一个 40 位的桶形移位器。有四条内部总线和两个地址产生器。此外，内部还集成了维特比加速器，用于提高维特比编译码的速度。先进的 DSP 结构可高效地实现无线通信系统中的各种功能。

（3）低功耗方式。TMS320C54x DSP 核可以在 3.3 V、2.5 V、1.8 V 甚至 1.2 V 的低电压下工作，三个低功耗方式（IDLE1、IDLE2 和 IDLE3）可以节省 DSP 的功耗，TMS320C54x 特别适合于无线移动设备。

（4）智能外设。除了标准的串行口和时分复用（TDM）串行口外，还提供了自动缓冲串行口 BSP（Buffered Serial Port）、多通道缓冲串行口 McBSP（Multichannel BSP）和与外部处理器通信的 HPI（Host Port Interface）接口。BSP 可提供数据缓冲的读写能力，从而可以降低处理器的额外开销，指令周期为 20 ns 时，BSP 的最大的数据吞吐量为 50 M 比特/秒，即使在 IDLE 方式下，BSP 也可以全速工作。McBSP 可与 128 个通道进行收发通信。HPI 可以与外部标准的微处理器直接接口。

图 2-14 是 TMS320C54x DSP 的方框图。

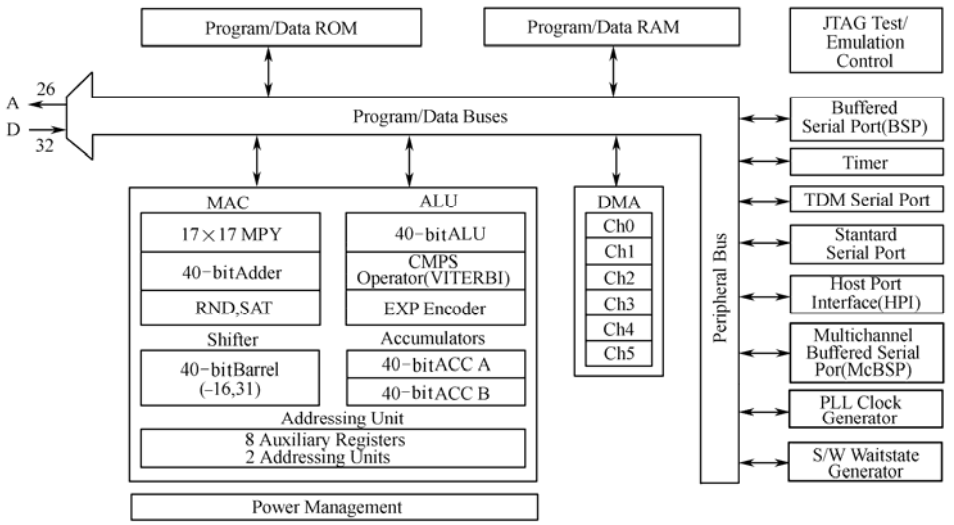


图 2-14　TMS320C54x DSP 的方框图

该系列的 TMS320VC5470/71 是一种多处理器芯片，它将 DSP 子系统和 RISC 微处理器（MCU）子系统集成在一个芯片上，其中 DSP 子系统基于 TMS320C54x 结构，而 RISC MCU 子系统则基于 ARM7TDMI 核。

DSP 子系统具有如下模块：

- TMS320C54x DSP 核；
- 72 K 字的内部 SRAM（32 K 字数据，40 K 程序）；
- ARM 口接口，支持 MCU 访问 8 K 字的 DSP 数据 SRAM；
- 二个多通道缓冲串行口（McBSP）；

- 锁相环（PLL）；
- 1 个定时器；
- 1 个 DMA 控制器；
- 可编程等待状态发生器；
- 外部存储器接口。

MCU 子系统包括如下模块：

- ARM7TDMI CPU 核（32/16 位 RISC 处理器），具有扩展仿真能力；
- MCU 存储器接口，包括外部 SRAM、Flash、ROM 和 SDRAM；
- 片上 16 K 字节（4 K×32）零等待 SRAM；
- MCU 通用 I/O（GPIO），支持 8×8 键盘；
- 三个定时器（两个通用定时器，一个 watchdog 定时器）；
- 兼容 IrDA 的 UART，支持 IrDA 模式和 UART 模式（无硬件流控）；
- MCU 子系统中断处理器；
- 时钟产生器；
- I^2C 接口；
- 串行外设接口；
- 锁相环。

TMS320C55x 是目前功耗最低的一种 DSP 芯片，与 TMS320C54x 软件兼容，与 TMS320C54x 相比，其综合性能提高了 5 倍，而功耗仅为 TMS320C54x 的 1/6。TMS320C55x 采用变指令长度以提高代码效率，增强并行机制以提高循环效率。TMS320C55x 以其极低的功耗和优越的性能可以在通信、消费电子等很多领域得到广泛的应用。

表 2-5 是 TMS320C5000 系列两类 DSP 芯片的主要特性比较。以下是 TMS320C55x 系列中的 TMS320C5510 芯片的主要性能：

表 2-5　TMS320C5000 两类 DSP 芯片的主要特性比较

特　　性		TMS320C54x　单核 DSP	TMS320C55x DSP
功耗（mW/MIPS）		0.54	0.05
MIPS 或 MMACS		30~532	140~800
代码密度			变指令长度结构
功能单元	MACs	1	2
	ALUs	1	2
	累加器	2	4
程序获取		16 位	32 位
指令长度		固定 16 位	8~48 位可变

- 6.25/5 ns 指令周期；
- 160/200 MHz 时钟速率；
- 双乘法器；
- 160 K×16 的片内 RAM；
- 16 K×16 的片内 ROM；
- 三个多通道缓冲串行口；

- 32 位外部存储器接口；
- 8/16 位增强型主机接口（HPI）；
- 超薄封装（240 引脚 BGA）；
- 6 通道 DMA 控制器。



2.7.4　TMS320C62x/C64x 系列

TMS320C62x 是 TI 公司于 1997 年开发的一种新型定点 DSP 芯片。该芯片的内部结构与以前的 DSP 不同，内部集成了多个功能单元，可同时执行 8 条指令，其运算能力可达 2400 MIPS。其主要特点有：

（1）运行速度快。指令周期最小为 3.3 ns，运算能力为 2400 MIPS。

（2）内部结构不同于一般 DSP，芯片内部同时集成了 2 个乘法器和 6 个算术运算单元，且它们之间是高度正交的，使得在一个指令周期内最大能支持 8 条 32 位的指令。

（3）指令集不同。为充分发挥其内部集成的各执行单元的独立运行能力，TI 公司使用了 VelociTI 超长指令字（VLIW）结构。它在一条指令中组合了几个执行单元，结合其独特的内部结构，可在一个时钟周期内并行执行多条指令。

（4）大容量的片内存储器和大范围的寻址能力。片内最多集成了 512 K 字节程序存储器和 512 K 字节数据存储器，并拥有 32 位的外部存储器界面。

（5）智能外设。内部集成了 4~16 个 DMA 接口，2~3 个多通道缓存串行口，两个 32 位计时器。

（6）低廉的使用成本。在一个无线基站的应用中，每片 TMS320C62xx 能同时完成 30 路的语音编解码，每路成本为 3 美元，而以前的 DSP 系列最大只能完成 5 路，每路的成本为 7 美元。

这种芯片适合于无线基站、无线 PDA、组合 Modem、GPS 导航等需要大运算能力的应用场合。

TMS320C64x 是 TMS320C6000 系列 DSP 中的最新的高性能定点芯片，其软件与 TMS320C62x 完全兼容。TMS320C64x 采用 VelociTI.2 结构的 DSP 核，增强的并行机制可以在单周期内完成四个 16×16 位或 8 个 8×8 位乘累加操作。采用两级缓冲存储器（Cache）机制，第一级中程序和数据各有 16 KB，而第二级中程序和数据共用 128 KB。增强的 32 通道 DMA 控制器具有高效的数据传输引擎，可以提供超过 2 Gb/s 的持续带宽。与 TMS320C62x 相比，TMS320C64x 的总体性能提高了 10 倍。表 2-6 是 TMS320C64x 与 TMS320C62x 的性能比较。

表 2-6　TMS320C64x 与 TMS320C62x 的性能比较

特　　性	TMS320C62x　DSP 核 Veloci TI 结构	TMS320C64x DSP 核 Veloci TI.2 结构	性能改善
MHz	150~300	600~1100	4 倍
MIPS	1 200~2 400	4 800~8 800	4 倍
16 位 MMACs	300~600	2 400~4 400	8 倍
8 位 MMACs	300~600	4 800~8 800	16 倍
通信	普通	特殊目的指令	8 倍
图像	普通	特殊目的指令	15 倍

2.8 TI 浮点 DSP 芯片

TI 公司的浮点 DSP 芯片包括 TMS320C3x、TMS320C4x、TMS320C8x、TMS320C67x 等。其中，TMS320C3x 是较早推出、应用较广的浮点芯片，性能价格比高，TI 后来又在该系列中增加了高性能、低功耗的 TMS320VC33。TMS320C4x 是 TMS320C3x 之后推出的高性能芯片，TMS320C8x 是将多个 TMS320C4x DSP 核集成在一起的 DSP 芯片。随着 TMS320C67x 系列浮点芯片的推出，TMS320C4x 和 TMS320C8x 已逐步退出市场。下面重点介绍 TMS320C3x 和 TMS320C67x 两种浮点芯片。

2.8.1 TMS320C3x

TMS320C3x 是 TI 的第三代产品，也是第一代浮点 DSP 芯片。TMS320C3x 中目前具有 TMS320C30、TMS320C31、TMS320C32 和 TMS320VC33 四种。TMS320C31 是 TMS320C30 的简化和改进型，它在 TMS320C30 的基础上减去了一般用户不常用的一些资源，降低了成本，是一个性能价格比较高的浮点处理器，在国内已得到了较广泛的应用。TMS320C32 是 TMS320C31 的进一步简化和改进，而 TMS320VC33 则是一种低功耗高性能的浮点芯片。表 2-7 示出了四种芯片的比较。

表 2-7 四种 TMS320C3x 芯片的比较

特　　征	TMS320C30	TMS320C31	TMS320C32	TMS320VC33
数据/程序总线	主总线 STRB： 32 位数据，24 位地址 扩展总线 IOSTRB： 32 位数据，13 位地址	STRB： 32 位数据，24 位地址	STRB0：8/16/32 位 STRB1：8/16/32 位 IOSTRB：32 位	STRB： 32 位数据，24 位地址
运行速度（最高）	50 MFLOPS/25 MIPS	80 MFLOPS/40 MIPS	60 MFLOPS/30 MIPS	150 MFLOPS/75 MIPS
内部 RAM	2 K 字	2 K 字	512 字	34 K 字
串行 I/O 口	二个串行口	一个串行口	一个串行口	一个串行口
用户程序、数据 ROM	4 K 字/16 K 字节	无	无	无
DMA 控制器	单通道	单通道	双通道	单通道
程序和数据 存储器宽度	32 位	32 位	程序 16/32 位可选 数据 8/16/32 位可选	32 位
外部中断触发	电平触发	电平触发	电平/边沿触发可选	电平/边沿触发可选
中断矢量表地址	固定	固定	用户可重定位	固定
程序引导	无	用户可选	用户可选	用户可选

2.8.1.1 TMS320C3x 的硬件资源

TMS320C30 的硬件资源相当丰富。其内部包含了 2 K×32 位的快速 RAM 块。分开的程序总线、数据总线和 DMA 总线使得取指、读写数据和 DMA 操作可并行进行，如 CPU 可以在一个周期内完成下列操作: 在一个 RAM 块中存取两个数据值，进行一次外部取指，DMA 装入到另一个 RAM 块。64×32 位的指令 Cache 用来存储经常使用的代码块，这可大大减少片

外访问的次数，从而提高程序运行速度。由于主总线的地址总线是 24 位，因此 TMS320C30 可以访问多达 16 M 的 32 位字的存储器空间，程序、数据和 I/O 空间都包含在 TMS320C30 这个 16 M 字的空间中。

TMS320C30 有一个 40/32 位的浮点/整数乘法器，这个乘法器可以在单周期内完成 24 位整数和 32 位浮点值的乘法，为了达到更高的性能，可以采用并行指令在单周期内完成一次乘法和 ALU 操作。当乘法器进行浮点乘法时，输入是 32 位单精度浮点值，而输出则是 40 位的扩展精度浮点数。整数乘法时，输入是 24 位，而输出则是 32 位。

TMS320C30 的整数/浮点算术逻辑运算单元 ALU 可以进行单周期的 32 位整数、32 位逻辑数和 40 位的浮点数的操作，ALU 的运算结果是 32 位整数或 40 位的浮点数。内部的桶形移位器可以在单周期内进行高达 32 位的左移或右移。内部总线 CPU1/CPU2 和 REG1/REG2 可从存储器中取两个操作数和从寄存器取两个操作数，这就使得对四个操作数进行的乘和加/减可以在单周期内完成。

TMS320C30 内有两个辅助寄存器算术单元（ARAU0 和 ARAU1），它们可以在单周期内产生两个地址。ARAU 的操作是和乘法器及 ALU 的操作并行进行的。ARAU 支持多种寻址方式。

TMS320C30 提供 28 个寄存器，这些寄存器可以由乘法器和 ALU 操作，可以用作通用的寄存器。但是，它们也有一些特殊的功能，如 8 个扩展精度的寄存器（R0~R7），可当累加器使用，特别适合用于存放扩展精度的浮点结果。8 个辅助寄存器（AR0~AR7）支持一系列间接寻址方式，并且可以用作通用的 32 位的整数和逻辑寄存器。其他的寄存器提供一些系统功能，如寻址、堆栈管理、处理器状态、中断和块重复等。

TMS320C30 的外设是通过存储器映射的寄存器对外设总线进行控制，外设总线由 32 位数据总线和 24 位地址总线组成，允许与外设直接通信。TMS320C30 的外设包括两个可支持 8/16/24/32 位数据交换的串行口和两个 32 位定时器。串行口的时钟可以由内部产生也可由外部提供，串行口的引脚可以配置为通用的 I/O 引脚，特殊的握手方式可保证 TMS320C30 与串行口的同步。定时器是通用的 32 位定时器或事件计数器，具有两种信号方式，可由内部或外部提供时钟，每个定时器对应有一个 I/O 引脚，可作为定时器的输入时钟或输出时钟，也可配置为通用的 I/O 引脚。

片内的 DMA 功能使得 CPU 与 I/O 操作可同时进行。DMA 控制器可以在存储器的任何地址进行读写操作而不干扰 CPU 的操作，因此 TMS320C30 可与外部的慢速存储器或外设接口而不降低 CPU 的吞吐量。

两个通用的外部引脚（XF0，XF1）可由软件设置为输入或输出，这两个引脚也用作 TMS320C30的互锁操作，用于多处理器之间的通信。

除了主总线之外，TMS320C30 还有一条扩展总线。扩展总线的数据线为 24 位，地址线为 13 位，其操作可与主总线操作同时进行，也可独立于主总线。与主总线相比，扩展总线上的 I/O 口的读写周期为 2 个 H1 周期，即是主总线的 2 倍。扩展总线通常用于 I/O 操作，如 A/D、D/A 等。

TMS320C30 内部有 4 K 字的 ROM，在微计算机方式下，程序由内部开始运行。

TMS320C31 是 TMS320C30 的简化和改进型，它去掉了扩展总线、一个串行口和内部的 4 K 字 ROM，增加了程序的引导功能。当 MCBL/$\overline{\text{MP}}$引脚为高电平时，工作于微计算机自引导方式，在这种方式下，TMS320C31 可从三个特定的地址装入程序（BOOT1，BOOT2，BOOT3），也可从串行口装入程序。与 EPROM 接口时，用户可用软件设定等待周期个数，当然也可用硬件来产

生等待周期，可设定是按字节（8 位）、半字（16 位）或字（32 位）装入，应用灵活。从串行口装入时，采用固定的 32 位突发方式。装入方式设置是通过复位之后设置四个外部中断之一来实现。

TMS320C32 是 TMS320C31 的简化和改进型。简化主要是将内部的 RAM 由 2 K 字减少为 512 字，以降低成本。改进之处包括：外部存储器宽度可变、中断矢量表可重定位、外部中断可由软件选择是电平触发还是沿触发、DMA 控制器增加了一个通道，此外，还增加了两种节电运行方式。

TMS320VC33 是 TMS320C3x 系列中性能最高、功耗最低的一种芯片。该芯片采用 3.3 V 电压（核心电压是 1.8 V），即使工作在最高速度，其功耗也低于 200 mW。内部集成了 34 K 字的双访问静态 RAM，可满足大部分系统对 RAM 的要求。芯片具有一个串行口、两个 32 位定时器、一个 DMA 控制器、两个外部标志输出 XF0 和 XF1。EDGEMODE 引脚可以选择外部中断是电平触发还是沿触发，芯片具有自引导功能。此外，还具有两种节电工作模式。

2.8.1.2 TMS320C3x 的软件资源

1．丰富的指令系统

TMS320C3x 的汇编语言指令集特别适合于数字信号处理。所有指令占一个机器字长，大部分指令是单周期的。指令集共有 113 条指令，可以分为六类: 数据传送类， 二操作数算术/逻辑类，三操作数算术/逻辑类，程序控制类，互锁操作类及并行操作类。12 条数据传送指令可从存储器中读一个字装入寄存器，将一个字从寄存器中存入存储器中及进行堆栈操作。二操作数指令有 35 条，提供整数、浮点、逻辑运算及多精度算术操作。17 条三操作数指令可以在一个指令周期内完成具有三个操作数的运算，其中二个是源操作数，另一个是目的操作数。程序控制指令共 16 条，它们影响程序的流向，其中有块重复指令 RPTB 和单指令重复指令 RPTS。除了有标准跳转指令外，还有延迟跳转指令，有些指令具有条件运算功能。5 条互锁操作指令主要用来进行多处理器之间的通信。剩下的 28 条指令都是并行操作指令，每条并行指令由两条指令用符号 || 连接，并行操作指令可使 || 前后的两条指令并行完成。需要注意的是，并不是任意两条指令都可构成并行指令。

2．灵活的程序控制

TMS320C3x 提供相当灵活的程序流控制。软件的程序控制包括重复、跳转、调用、陷阱及返回等。

TMS320C3x 重复方式可以实现无开销循环。块重复指令 RPTB 可以使一块代码重复执行指定的次数，而单指令重复 RPTS 使一条指令重复执行指定的次数，由于 RPTS 取指次数只有一次，因此减少了总线冲突的机会。这里需要特别注意的是，RPTS 指令禁止中断，而 RPTB 无此禁忌。因此在需要中断的场合，建议采用 RPTB 指令代替 RPTS 指令。

TMS320C3x 提供两种类型的跳转：标准跳转和延迟跳转。标准跳转首先将流水线清空然后执行跳转，这可保证程序计数器的正确管理，但却使跳转指令需四个周期才能执行完毕。延迟跳转则相反，它并不将流水线清空，而是保证在程序跳转之前执行它下面的三条指令，这样使得跳转指令本身仅需一个指令周期。当然有些指令（如 CALL，RPTB 等）不能出现在延迟跳转指令紧邻的三条指令中。延迟跳转指令在之后的三条指令完成前禁止中断。

互锁操作可用来实现多个 TMS320C3x 之间的通信。通过采用外部信号（XF0，XF1），互

锁操作指令提供强有力的同步机制，可用来实现忙等待循环、两个 TMS320C3x 之间的同步（无需额外的硬件支撑）和多个计数器控制等。

3．完善的寻址方式

TMS320C3x 支持 5 类寻址方式，在这些类中又可采用 6 种寻址类型。5 类寻址方式是：①通用寻址方式；②三操作数寻址方式；③并行寻址方式；④长立即数寻址方式；⑤条件跳转寻址方式。6 种寻址类型是：①寄存器寻址；②直接寻址；③间接寻址；④短立即数寻址；⑤长立即数寻址；⑥相对寻址。此外，TMS320C30 还有两种数字信号处理中非常有用的寻址方式，一种是循环寻址方式，主要用于卷积和相关等运算中存储器循环缓冲；另一种寻址方式是比特反转寻址，主要用于 FFT 运算。

2.8.2　TMS320C67x

这是 TI 公司继定点 DSP 芯片 TMS320C62x 系列后开发的一种新型浮点 DSP 芯片。该芯片的内部结构在 TMS320C62x 的基础上加以改进，内部同样集成了多个功能单元，可同时执行 8 条指令，其运算能力可达 1 G FLOPS。除了具有 TMS320C62x 系列的特点外，其主要特点还有：

（1）运行速度快。指令周期为 6 ns，峰值运算能力为 1336 MIPS，对于单精度运算可达 1 G FLOPS，对于双精度运算可达 250 M FLOPS。

（2）硬件支持 IEEE 格式的 32 比特 单精度与 64 比特双精度浮点操作。

（3）集成了 32×32 比特的乘法器，其结果可为 32 或 64 比特。

（4）TMS320C67x 的指令集在 TMS320C62x 的指令集基础上增加了浮点执行能力，可以看作是 TMS320C62x 指令集的超集。TMS320C62x 指令能在 TMS320C67x 上运行，而无需任何改变。

与 TMS320C62x 系列芯片一样，由于其出色的运算能力、高效的指令集、智能外设、大容量的片内存储器和大范围的寻址能力，这个系列的芯片适合于对运算能力和存储量有高要求的应用场合。

2.9　其他 DSP 芯片简介

2.9.1　AD 公司的 DSP 芯片

美国 AD 公司的在 DSP 芯片市场上也占有一定的份额。与 TI 公司相比，AD 公司的 DSP 芯片有自己的特点，如系统时钟一般不经分频直接使用，串行口带有硬件压扩，可从 8 位 EPROM 引导程序，可编程等待状态发生器等。

AD 公司的 DSP 芯片可以分为定点 DSP 芯片和浮点 DSP 芯片两大类。ADSP21XX 系列为定点 DSP 芯片，ADSP21XXX 系列为浮点 DSP 芯片。目前，定点 DSP 芯片主要有 ADSP2101/2103/2105，ASDP2111/2115，ADSP2161/2162/2163/2164/2165/2166 以及 ADSP2171/2173/2181 等。浮点 DSP 芯片主要有 ADSP21000/21020，ADSP21060/21062 等。

AD 公司的定点 DSP 芯片的程序字长为 24 位，数据字长为 16 位。运算速度较快，内部具有较为丰富的硬件资源，一般具有 2 个串行口、一个内部定时器和三个以上的外部中断源，此外还提供 8 位 EPROM 程序引导方式。具有一套高效的指令集，如无开销循环、多功能指令、条件执行等。

ADSP2101 的指令周期有 80 ns、60 ns 和 50 ns 三种，内部有 2 K 字的程序 RAM 和 1 K 字的数据 RAM。ADSP2103 与 ADSP2101 相比，指令周期为 100 ns，工作电压为 3.3 V。ADSP2105 是 ADSP2101 的简化，指令周期为 72 ns，内部的程序 RAM 为 1K 字，数据 RAM 为 512 字，串行口减为一个。

ADSP216X 系列的指令周期为 50 ns 至 100 ns，与其他定点芯片相比，具有较大的内部程序 ROM，如 ADSP2161/2163 内部提供了 8 K 的程序 ROM，ADSP2162/2164 内部提供 4K 程序 ROM，工作电压为 3.3 V，这些芯片的内部数据 RAM 均为 512 字。而 ADSP2165/2166 除了具有 1K 字的程序 ROM 外，还提供了 12 K 字的程序 RAM 和 4 K 字的数据 RAM，其中 ADSP2166 的工作电压为 3.3 V。

ADSP2171 的指令周期为 30 ns，速度达 33.3 MIPS，是 AD 公司 DSP 芯片中运算速度最快的定点芯片之一。内部具有 2 K 字的程序 RAM 和 2 K 字的数据 RAM。ADSP2173 的资源与 ADSP2171 相同，工作电压为 3.3 V。

目前 ADSP 的定点 DSP 芯片中，处理能力最强的当数 ADSP2181。这种芯片具有以下几个特点：

（1）运算速度快。指令周期为 30 ns，运算能力为 33.3 MIPS。

（2）片内空间大。内部的程序和数据 RAM 均为 16 K 字，共 80 KB。

（3）数据交换速度快。内部具有数据直接存储传输接口（IDMA），便于其他处理器高速存储 ADSP2181 的片内 RAM。为了扩展数据存储能力，设计了字节数据直接存储传输接口（BDMA），最大可以扩展到 4 M 字节。两个串行口都具有自动数据缓冲功能，并且支持 DMA 传输。

（4）支持 8 位 EPROM 和通过 IDMA 方式的程序引导。

（5）如果采用基 4 FFT 做 1024 点复数 FFT 运算，运算时间仅为 1.07 ms。

ADSP2181 在一个处理器周期内可以完成以下功能：

- 产生下一个程序地址；
- 取下一个指令；
- 进行一个或二个数据移动；
- 更新一个或二个数据地址指针；
- 进行一次数据运算。

与此同时，还可从两个串行口发送或接收数据、通过 IDMA 或 BDMA 发送或接收数据以及内部定时器计数。

AD 公司的浮点 DSP 芯片包括 ADSP21020、21060 和 21062 等，程序存储器为 48 位，数据存储器为 40 位，支持 32 位单精度和 40 位扩展精度的 IEEE 浮点格式，内部具有 32×48 位的程序 Cache，有 3~4 个外部中断源。

ADSP21060 采用超级的哈佛结构，具有四条独立的总线（两条数据总线、一条程序总线和一条 I/O 总线），内部集成了大容量的 SRAM 和专用 I/O 总线支持的外设，指令周期为 25 ns，是一个高性能的浮点 DSP 芯片。其主要特点包括：

（1）运算速度达 40 MIPS 和 80 MFLOPS，最高达 120 MFLOPS。每条指令均在一个周期内完成。

（2）片内具有 4 M 位的 SRAM，可灵活地进行配置，如配置为 128 K 字的数据存储器（32 位）和 80 K 字的程序存储器（48 位）。可寻址 4 G 字的外部存储器。

（3）具有取模和比特反转的双数据地址产生器。

（4）10 个 DMA 通道。六个点到点连接口，传输速率为 240 MB/s。

（5）支持多处理器连接，提供与 16/32 位微处理器的接口。外部微处理器可直接读写内部 RAM。

（6）两个具有μ/A 律压扩功能的同步串行口。

（7）支持可编程等待状态发生，可用 8 位 EPROM 或外部处理器引导程序。

（8）1024 点复数 FFT 的运算时间为 0.46 ms。

（9）支持 IEEE JTAG 1149.1 标准仿真接口。

2.9.2　AT&T 公司 DSP 芯片

AT&T 是第一家推出高性能浮点 DSP 芯片的公司。AT&T 公司的 DSP 芯片包括定点和浮点两大类。定点 DSP 芯片中有代表性的主要包括 DSP16、DSP16A、DSP16C、DSP1610 和 DSP1616 等。浮点 DSP 芯片中比较有代表性的包括 DSP32、DSP32C 和 DSP3210 等。

AT&T 定点 DSP 芯片的程序和数据字长均为 16 位，有两个精度为 36 位的累加器，具有一个深度为 15 字的指令 Cache，支持最多 127 次的无开销循环。DSP16 的指令周期为 55 ns 和 75 ns，累加器长度为 36 位，片内有 2 K 字的程序 ROM 和 512 字的数据 RAM。DSP16A 速度最快的版本为 25 ns 的指令周期，片内有 12 K 字的程序 ROM 和 2 K 字的数据 RAM。DSP16C 的指令周期为 38.5 ns 和 76.9 ns，片内存储器资源与 DSP16A 相同，增加了片内的 Codec，此外，还有 1 个 4 引脚的 JTAG 仿真口。DSP1610 片内有 512 字的引导 ROM 和 8 K 字的双口 RAM，支持硬件和软件等待状态。DSP1616 片内有 12 K 字的 ROM 和 2 K 字的双口 RAM，支持软件等待状态。DSP1610 和 1616 提供了仿真接口。

DSP32C 是 DSP32 的增强型，是性能较优的一种浮点 DSP 芯片。其主要特点包括：

（1）80/100 ns 的指令周期。

（2）地址和数据总线可以在单个指令周期内访问 4 次。

（3）片内具有 3 个 512 字的 RAM 块，或 2 个 512 字的 RAM 块加 1 个 4 K 字的 ROM 块，可以寻址 4 M 字的外部存储器。

（4）具有串行和并行 I/O 接口。串行 I/O 采用双缓冲，支持 8/16/24/32 位串行数据传输，外部微处理器可以控制 DSP32C 的 8/16 位并行口。

（5）采用专用的浮点格式，可在单周期内与 IEEE-754 浮点格式进行转换。

（6）具有 4 个 40 位精度的累加器和 22 个通用寄存器。

（7）支持无开销循环和硬件等待状态。

DSP3210 内部具有 2 个 1 K 字的 RAM 块和 512 字的引导 ROM，外部寻址空间达 4 G 字节，可以用软件编程产生等待状态，具有串行口、定时器、DMA 控制器和一个与 Motorola 和 Intel 微处理器兼容的 32 位总线接口。

2.9.3　Motorola 公司 DSP 芯片

Motorola 公司的 DSP 芯片可分为定点、浮点和专用三种。

定点 DSP 芯片以 MC56000、MC56001、MC56002 为代表。程序和数据字长为 24 位，有两个精度为 56 位的累加器。DSP56001 的指令周期为 60 ns 和 74 ns 两种。片内具有 512 字的程序 RAM、512 字的数据 RAM 和 512 字的数据 ROM。三个分开的存储器空间，每个均可寻

址 64 K 字。片内 32 字的引导程序可以从外部 EPROM 装入程序。支持 8 位异步和 80~24 位同步串行 I/O 接口。并行接口可与外部微处理器接口，支持硬件和软件等待状态产生。MC56000 是 ROM 型的 DSP 芯片，内部具有 2 K 字的程序 ROM。MC56002 则是一个低功耗型芯片，可以在 2.0~5.5 V 电压范围内工作。

浮点 DSP 芯片以 MC96002 为代表，采用 IEEE-754 标准浮点格式，累加器精度达 96 位，可支持双精度浮点数。该芯片的指令周期为 50/60/74 ns。片内有 3 个 32 位地址总线和 5 个 32 位数据总线。内部具有 1 K 字的程序 RAM、1 K 字的数据 RAM 和 1 K 字的数据 ROM。64 字的引导 ROM 可以从外部 8 位 EPROM 引导程序。内部具有 10 个 96 位或 30 个 32 位基于寄存器的累加器。支持无开销循环、硬件和软件等待状态产生。具有三个独立的存储空间，每个空间可寻址 4 G 字。

MC56200 是一种基于 MC56001 DSP 核，适合于自适应滤波的专用定点 DSP 芯片，指令周期为 97.5 ns，程序字长和数据字长分别为 24 位和 16 位。内部的程序和数据 RAM 均为 256 字，累加器精度为 40 位。MC56156 则是一个在片内集成了过取样∑－Δ话带 Codec 模数转换器和锁相环的 DSP 芯片，主要用于蜂窝电话等通信应用，其指令周期为 33/50 ns。

除了以上介绍的一些 DSP 芯片之外，还有一些公司的 DSP 芯片也较著名。例如 NEC 公司的μPD77C25、μPD77220 定点 DSP 芯片和μPD77240 浮点 DSP 芯片等。

2.10 小　　结

本章首先介绍了 DSP 芯片的基本结构，随后比较详细地介绍了 TI 公司的系列 DSP 芯片的基本特征，并简要介绍了 AD 等公司的 DSP 芯片。了解 DSP 芯片的结构和特征是采用 DSP 芯片设计 DSP 系统的基础。需要特别指出的是，由于 DSP 芯片的发展速度很快，用户在选用 DSP 芯片时，必须根据市场行情选用生产厂家主推的产品。如 TI 公司，目前比较流行的定点 DSP 芯片是 TMS320C2xx、TMS320C54x/C55x、TMS320C62x 等，即使是同一系列的 DSP 芯片，如 TMS320C54x，系列中的某些芯片也不再建议使用。

2.11 习题与思考题

（1）DSP 芯片的典型结构是怎样的？
（2）ALU 与累加器的区别是什么？
（3）什么是冯·诺伊曼结构和哈佛结构？两者有什么区别？
（4）什么是流水线操作？简述其工作原理。
（5）如何利用通用 I/O 口实现 UART 通信？
（6）HPI 口的特点是什么？
（7）标准同步串行口应包括哪些基本信号？
（8）软件可编程等待状态产生器的作用是什么？
（9）TI 公司的定点 DSP 芯片有哪些？各有什么特点？
（10）TI 公司的浮点 DSP 芯片有哪些？各有什么特点？
（11）除 TI 公司外，比较著名的 DSP 芯片生产商有哪些？

第 3 章　DSP 的数值运算基础

3.1　引　　言

DSP 芯片的主要功能是快速进行数值运算。DSP 芯片包括定点芯片和浮点芯片两大类，其中，定点芯片的操作数一般是整型数，而浮点芯片的操作数可以是整型数，也可以是小数。那么，定点 DSP 芯片能不能处理小数呢？如何将一个浮点数转换为定点数？如何将一个基于浮点数编写的 C 程序转换为定点数程序？对于非线性函数，如何在定点 DSP 芯片上实现？此外，浮点 DSP 芯片中数值是如何表示的？这些问题将在本章予以回答。

3.2　定点的基本概念

3.2.1　数的定标

在定点 DSP 芯片中，采用定点数进行数值运算，其操作数一般采用整型数来表示。一个整型数的最大表示范围取决于 DSP 芯片所给定的字长，一般为 16 位或 24 位。显然，字长越长，所能表示的数的范围越大，精度也越高。本书以 16 位字长为例进行介绍。

DSP 芯片的操作数以 2 的补码形式表示。每个 16 位数用一个符号位来表示数的正负（0 表示数值为正，1 则表示数值为负），其余 15 位表示数值的大小。因此，

二进制数 0010000000000011B = 8195

二进制数 1111111111111100B = –4

对 DSP 芯片而言，参与数值运算的数就是 16 位的整型数。但在许多情况下，数学运算过程中的数不一定都是整数。那么，如何让定点 DSP 芯片处理小数呢？这其中的关键就是由程序员来确定一个数的“小数点”处于 16 位中的哪一位。这就是数的定标。

通过设定“小数点”在 16 位数中的不同位置，就可以表示不同大小和不同精度的小数了。数的定标通常有 *Q* 表示法和 *S* 表示法两种。表 3-1 列出了一个 16 位有符号数的 16 种 *Q* 表示、*S* 表示、精度及它们所能表示的十进制数值范围。必须指出的是，这里所说的“小数点”是一个隐式的小数点，对 DSP 芯片而言，参与运算的就是 16 位的整型数，并不存在真正意义上的小数点。

一个 16 位有符号数，最左边的一位是符号位，其余 15 位包括整数位和小数位。对于 *Q* 表示法，*Q* 后面的数表示该数的小数点右边有几位，如 Q15 表示该数的小数点右边有 15 位，即有 15 位小数，没有整数位；Q10 表示该数的小数点右边有 10 位，即有 10 位小数和 5 位整数位。对于 *S* 表示法，它清楚地表示出了该数小数点左右的位数，如 S1.14 表示该数小数点左边有 1 位，即有 1 位整数，而小数点右边有 14 位，即有 14 位小数。

从表 3-1 可以看出，同样一个 16 位数，若小数点设定的位置不同，它所表示的数也就不同。

表 3-1　*Q* 表示、*S* 表示、精度及数值范围

Q 表示	*S* 表示	精度	十进制数表示范围
Q15	S0.15	1/32768	−1≤*X*≤0.9999695
Q14	S1.14	1/16384	−2≤*X*≤1.9999390
Q13	S2.13	1/8192	−4≤*X*≤3.9998779
Q12	S3.12	1/4096	−8≤*X*≤7.9997559
Q11	S4.11	1/2048	−16≤*X*≤15.9995117
Q10	S5.10	1/1024	−32≤*X*≤31.9990234
Q9	S6.9	1/512	−64≤*X*≤63.9980469
Q8	S7.8	1/256	−128≤*X*≤127.9960938
Q7	S8.7	1/128	−256≤*X*≤255.9921875
Q6	S9.6	1/64	−512≤*X*≤511.9804375
Q5	S10.5	1/32	−1024≤*X*≤1023.96875
Q4	S11.4	1/16	−2048≤*X*≤2047.9375
Q3	S12.3	1/8	−4096≤*X*≤4095.875
Q2	S13.2	1/4	−8192≤*X*≤8191.75
Q1	S14.1	1/2	−16384≤*X*≤16383.5
Q0	S15.0	1	−32768≤*X*≤32767

例如：

十六进制数 2000H = 8192，用 Q0 表示

十六进制数 2000H = 0.25，用 Q15 表示

但对于 DSP 芯片来说，处理方法是完全相同的。

从表 3-1 还可以看出，不同的 *Q* 所表示的数不仅范围不同，而且精度也不相同。*Q* 越大，表示的数值范围越小，但精度越高（量化步长越小）；相反，*Q* 越小，表示的数值范围越大，但精度就越低（量化步长越大）。不同 *Q* 值所对应的正最大值、负最大值和量化步长可表示为：

正最大值：$2^{(15-Q)}-2^{-Q}=(2^{15}-1)\times 2^{-Q}$

负最大值：$-2^{(15-Q)}=-2^{15}\times 2^{-Q}$

精　　度：2^{-Q}

例如，Q0 的数值范围是–32768~＋32767，其精度为 1；而 Q15 的数值范围为–1~0.9999695，精度为 1/32768 = 0.00003051。显然，对定点数而言，数值表示范围与精度是一对矛盾，一个变量要想能够表示比较大的数值范围，必须以牺牲精度为代价；而想精度提高，则数的表示范围就相应地减小。在实际的定点算法中，为了达到最佳的性能，需要充分考虑到这一点。



3.2.2　数的转换

1．浮点数与定点数之间的转换

浮点数与定点数的转换关系可表示为：

浮点数（*x*）转换为定点数（x_q）：$x_q=(\text{int})x*2^Q$　　（3.1）

定点数（x_q）转换为浮点数（*x*）：$x=(\text{float})x_q*2^{-Q}$　　（3.2）

例如，浮点数 $x=0.6$，定标 $Q=15$，则定点数 $x_q=\lfloor 0.6\times 32768\rfloor=19660$，式中$\lfloor\ \rfloor$表示下取整。反之，一个用 $Q=15$ 表示的定点数 19660，转换为浮点数是 19660×2^{-15} = 19660/32768 = 0.599975585，显然，用定点数来表示 0.6 这个数存在一定的误差。

为了最大限度地保持数的精度，在将浮点数转换为定点数时，可以采取“四舍五入”的方法，即在取整运算前，先加上 0.5：

浮点数（x）转换为定点数（x_q）：$x_q = (\text{int})[x * 2^Q + 0.5]$　　（3.3）

同样，对于 0.6 这个数，采用上述方法转换得到的定点数将为 $\lfloor 0.6 \times 32768 + 0.5 \rfloor = 19661$，其浮点数为 $19661 \times 2^{-15} = 19661/32768 = 0.600006103$，显然其精度得到了提高。

2．定点二进制数转换为十进制数

对于一个给定 Q 值的二进制数，确定该数的等值十进制数大小有以下两种方法：

（1）先求整型数，然后利用式（3.2）进行计算

首先将该数视为一个整型数，计算该整型数的等值十进制数值，然后利用式（3.2）求该数。例如，设一个 Q=15 的二进制有符号数为 0100110011001100，由于左边第一位是符号位且为 0，因此它是一个正数，则其对应的十进制整型数值为：

$$2^2 + 2^3 + 2^6 + 2^7 + 2^{10} + 2^{11} + 2^{14} = 19660$$

故该数的等值十进制数为：

$$19660 / 2^{15} = 19660 / 32768 = 0.599975585$$

（2）根据 Q 值，确定小数点位置，然后进行计算

根据 Q 值确定小数点的位置，然后以小数点为中心，依次向左向右可得到每一位的权值大小，具体如下：

依次向左，各位的权值大小分别为：2^0、2^1、2^2、2^3、…，符号位除外；

依次向右，各位的权值大小分别为：2^{-1}、2^{-2}、2^{-3}、2^{-4}、…。

仍以二进制有符号数 0100110011001100 为例，若 Q=15，则该数相当于：

0.100110011001100

即隐含的小数点在符号位之后，由于该数的符号位为 0，因此它是一个正数，其等值十进制大小为：

$$2^{-1} + 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + 2^{-12} + 2^{-13} = 0.599975585$$

同样的二进制有符号数，若 Q=14，则该数相当于：

01.00110011001100

其等值十进制大小为：

$$2^0 + 2^{-3} + 2^{-4} + 2^{-7} + 2^{-8} + 2^{-11} + 2^{-12} = 1.199951172$$

需要注意的是，如果一个二进制数是负数，则首先应将它转换为无符号的二进制格式，然后利用上述方法进行计算。



3.2.3　溢出保护

由于定点数的表示范围是一定的，因此在进行定点数的加法或减法运算时，其结果就有可能出现超过数值表示范围的情况，这种现象称为溢出。

在进行定点运算时，必须考虑溢出的处理方法。因为，如果忽视溢出情况，就有可能导致灾难性的后果。

【例 3.1】两个 16 位有符号数 x 和 y 相加，结果也用 16 位有符号数表示，则：

x = 32766d = 0111111111111110b;

y = 3d = 0000000000000011b;

```
  0111111111111110    =32766
+ 0000000000000011    =3
------------------
  1000000000000001    =－32767
```

显然，x 与 y 相加的结果应该是 32769，但由于已经超过了表示范围，在不采取溢出保护措施的情况下，其结果却变成了–32767。

为了避免这种情况的发生，一般在 DSP 芯片中可以设置溢出保护功能。在溢出保护功能设置后，当发生溢出时，DSP 芯片自动将结果饱和设置为正最大值或负最大值。

如果设置了溢出保护功能，则上述加法的结果为 +32767，从而避免结果从 +32769 到 –32767 的灾难性后果。

在 DSP 芯片中，通常有专门的一个状态标志位来表示溢出模式。例如，对于 TMS320C54x，在状态寄存器 1（ST1）中有 1 位 OVM，采用下面的指令就可以设置是否进行溢出保护：

```
SSBX    OVM             ; OVM=1，设置溢出保护模式
RSBX    OVM             ; OVM=0，设置溢出不保护模式
```

TMS320C54x 中，发生正溢出的饱和值为 0x7FFF，即+32767，发生负溢出的饱和值为 0x8001，即−32767。

3.2.4 符号扩展

当一个字长小的数放到一个字长大的累加器进行运算时，必须注意数的符号问题。先来看下面的一个例子。

一个用 4 位表示的数 x，设 x 为 1011，即为十进制数−5，如果将它直接放到一个已经清 0 的 8 位字长累加器的低 4 位（符号不扩展），则得到 00001011，显然它表示+11，不是−5。如果在装载时进行符号扩展，即将累加器的高四位根据所存放数据的符号进行填充，得到 11111011，该数表示的数值为正确数−5。

在 DSP 芯片中进行二进制补码的运算时，通常参与运算的数为 16 位，而 DSP 累加器为 32 位，因此，为了使得运算能够得到正确的结果，需要把数的符号位扩展到累加器左边的高位。这就是 DSP 芯片中的符号扩展。

在 DSP 芯片中，通常有专门的一个标志位来表示符号扩展模式。例如，对于 TMS320C54x，在状态寄存器 1（ST1）中有 1 位 SXM，采用下面的指令就可以设置符号是否扩展：

```
SSBX    SXM             ; SXM=1，设置符号扩展模式
RSBX    SXM             ; SXM=0，设置符号不扩展模式
```

3.2.5 舍入与截尾

下面以取整运算来说明舍入（rounding）和截尾（truncation）的概念。

对一个数 x 进行取整处理，一般有以下两种处理方法：

（1）直接将小数部分去除，这就是截尾处理方法，也称下取整。

（2）将该数加 0.5，之后再将小数部分去除，这就是舍入处理方法，即通常所说的四舍五入。这种处理方法也称上取整。

【例 3.2】已知 x=123.3，y=123.7，试分别对 x、y 进行舍入和截尾处理。

对 x 进行舍入：round（x）=round（123.3）=trunc（123.3+0.5）=123；

对 x 进行截尾：trunc（x）=trunc（123.3）=123；

对 y 进行舍入：round（y）=round（123.7）=trunc（123.7+0.5）=124；

对 y 进行截尾：trunc（y）=trunc（123.7）=123。

从上述的例子可以看出，由于数 x 的小数部分为 0.3，采用舍入运算后得到的结果与截尾运算结果相同。而数 *y* 的小数部分为 0.7，采用舍入运算后得到的结果与截尾运算结果不同，数值精度要高一些。

对于 DSP 芯片的乘法运算，常需要用到舍入处理方法。下面我们举例予以说明。

设两个 *Q* 值均为 15 的 16 位有符号数 x 和 y 进行乘法运算，结果也采用 Q15 表示的 16 位数。我们将一般的运算过程描述如下：

（1）x 与 y 相乘，结果放在 32 位累加器中。由于 x 与 y 的 *Q* 值均为 15，因此，此时累加器中结果为 32 位，*Q* 值为 30。

（2）将累加器的数左移一位。此时累加器中的数的 *Q* 值变为 31。

（3）为了将结果表示为 16 位，一种方法是直接将低 16 位截尾，保留高 16 位，作为乘法结果。另一种方法就是在截尾之前进行舍入处理。方法就是在低 16 位数的最高位加 1（相当于加 32768），然后将低 16 位去除，将高 16 位作为乘法结果。

显然，进行舍入运算后，能够最大程度地保持 16 位结果的精度。

3.3　定点运算实现的基本原理

在编写 DSP 模拟算法时，为了方便，一般都是采用高级语言（如 C 语言）来编写模拟程序。程序中所用的变量一般既有整型数，又有浮点数。如例 3.3 程序中的变量 i 是整型数，而 pi 是浮点数，hamwindow 则是浮点数组。如无特别说明，下文中的“int”定义的是 16 位整型数，“long”定义的是 32 位整型数。

【例 3.3】256 点汉明窗计算

```
int    i;
float  pi=3.14159;
float  hamwindow[256];
for（i=0;i<256;i++）    hamwindow[i]=0.54-0.46*cos（2.0*pi*i/255）;
```

如果要将上述程序用某种定点 DSP 芯片来实现，则需将上述程序改写为 DSP 芯片的汇编语言程序。为了 DSP 程序调试的方便及模拟定点 DSP 实现时的算法性能，在编写 DSP 汇编程序之前一般需将高级语言浮点算法改写为高级语言定点算法，以模拟 DSP 定点实现时的性能。下面来讨论基本算术运算的定点实现方法。

3.3.1　加法/减法运算的 C 语言定点模拟

设浮点加法运算的表达式为：

```
float x,y,z;
z=x+y;
```

将浮点加法/减法转化为定点加法/减法时最重要的一点就是必须保证两个操作数的定标值一样。若两者不一样，则在做加法/减法运算前须先进行小数点的调整。为保证运算精度，需使 Q 值小的数调整为与另一个数的 Q 值一样大。此外，在做加法/减法运算时，必须注意结果可能会超过 16 位表示。如果加法/减法的结果超出 16 位的表示范围，则必须保留 32 位结果，以保持运算的精度。

1．结果不超过 16 位表示范围

设 x 的 Q 值为 Q_x，y 的 Q 值为 Q_y，且 $Q_x>Q_y$，加法/减法结果 z 的定标值为 Q_z，则

$$z=x+y \quad =>$$

$$z_q \cdot 2^{-Q_z} = x_q \cdot 2^{-Q_x} + y_q \cdot 2^{-Q_y}$$

$$= x_q \cdot 2^{-Q_x} + y_q \cdot 2^{(Q_x-Q_y)} \cdot 2^{-Q_x}$$

$$= [x_q + y_q \cdot 2^{(Q_x-Q_y)}] \cdot 2^{-Q_x} \quad =>$$

$$z_q = [x_q + y_q \cdot 2^{(Q_x-Q_y)}] \cdot 2^{(Q_z-Q_x)}$$

所以定点加法可以描述为：

```
int   x,y,z;
long  temp      /*临时变量*/
temp = y<<（Qx−Qy）;
temp = x＋temp;
z =（int）（temp>>（Qx−Qz））, 若 Qx≥Qz
z =（int）（temp<<（Qz−Qx））, 若 Qx≤Qz
```

【例 3.4】定点加法

设 $x = 0.5$，$y = 3.1$，则浮点运算结果为 $z = x+y = 0.5+3.1 = 3.6$;

$Q_x = 15$，$Q_y = 13$，$Q_z = 13$，则定点加法为：

x = 16384；y = 25395;

```
temp = 25395<<2 = 101580;
temp = x+temp = 16384+101580 = 117964;
z =（int）（117964L>>2）= 29491;
```

因为 z 的 Q 值为 13，所以定点值 $z = 29491$ 即为浮点值 $z = 29491/8192 \approx 3.6$。

【例 3.5】定点减法

设 x = 3.0，y = 3.1，则浮点运算结果为 z = x−y = 3.0−3.1 = −0.1;

$Q_x = 13$，$Q_y = 13$，$Q_z = 15$，则定点减法为：

x = 24576；y = 25295；

```
temp = 25395;
temp = x－temp = 24576－25395 = －819;
```

因为 $Q_x<Q_z$，故 z =（int）（−819<<2）= −3276。由于 z 的 Q 值为 15，所以定点值 z = −3276 即为浮点值 z = −3276/32768 ≈ −0.1。

2．结果超过 16 位表示范围

设 x 的 Q 值为 Q_x，y 的 Q 值为 Q_y，且 $Q_x>Q_y$，加法结果 z 的定标值为 Q_z，则定点加法为：

int x，y；

long temp，z；

temp = y<<（Q_x−Q_y）；

temp = x＋temp;

z = temp>>（Q_x−Q_z），若 $Q_x \geqslant Q_z$

z = temp<<（Q_z−Q_x），若 $Q_x \leqslant Q_z$

【例 3.6】结果超过 16 位的定点加法

设 $x = 15000$，$y = 20000$，则浮点运算值为 $z = x + y = 35000$，显然 $z > 32767$，因此 $Q_x = 1$，$Q_y = 0$，$Q_z = 0$，则定点加法为：

x = 30000；y = 20000；

temp = 20000<<1 = 40000;

temp = temp+x = 40000+30000 = 70000;

z = 70000L>>1 = 35000;

因为 z 的 Q 值为 0，所以定点值 z=35000 就是浮点值，这里 z 是一个长整型数。

当加法或加法的结果超过 16 位表示范围时，如果程序员事先能够了解到这种情况，并且需要保持运算精度时，则必须保持 32 位结果。如果程序中是按照 16 位数进行运算的，则超过 16 位实际上就是出现了溢出。如果不采取适当的措施，则数据溢出会导致运算精度的严重恶化。



3.3.2　乘法运算的 C 语言定点模拟

设浮点乘法运算的表达式为：

float x,y,z;

z = xy;

假设 x 的定标值为 Q_x，y 的定标值为 Q_y，乘积 z 的定标值为 Q_z，则

$$z = xy \Rightarrow$$

$$z_q \cdot 2^{-Q_z} = x_q \cdot y_q \cdot 2^{-(Q_x+Q_y)} \Rightarrow$$

$$z_q = (x_q y_q) 2^{Q_z-(Q_x+Q_y)}$$

所以定点表示的乘法为：

int x,y,z;

long temp;

temp = （long）x;

z = （temp×y） >> （Q_x+Q_y−Q_z）;

【例 3.7】定点乘法

设 x = 18.4，y = 36.8，则浮点运算结果为 =18.4×36.8 = 677.12;

根据上节，可设定 Q_x= 10，Q_y = 9，Q_z= 5，所以

x = 18841；y = 18841；

temp = 18841L;

z =（18841L*18841）>>（10+9−5）= 354983281L>>14 = 21666;

因为 z 的定标值为 5，故定点 z = 21666，即为浮点的 z = 21666/32 = 677.06。

3.3.3 除法运算的 C 语言定点模拟

设浮点除法运算的表达式为：

```
float x,y,z;
z = x/y;
```

假设被除数 x 的定标值为 Q_x，除数 y 的定标值为 Q_y，商 z 的定标值为 Q_z，则：

$$z = x/y \Rightarrow$$

$$z_q \cdot 2^{-Q_z} = \frac{x_q \cdot 2^{-Q_x}}{y_q \cdot 2^{-Q_y}} \Rightarrow$$

$$z_q = \frac{x_q \cdot 2^{(Q_z - Q_x + Q_y)}}{y_q}$$

所以定点表示的除法为：

```
int x,y,z;
long temp;
temp =（long）x;
z =（temp<<（Qz−Qx+Qy））/y;
```

【例 3.8】定点除法

设 x = 18.4，y = 36.8，浮点运算值为 z = x/y = 18.4/36.8 = 0.5;

根据上节，可设定 Q_x= 10，Q_y= 9，Q_z = 15；所以有

```
x = 18841, y = 18841;
temp = （long）18841;
z =（18841L<<（15−10+9）/18841 = 308690944L/18841 = 16384;
```

因为商 z 的定标值为 15，所以定点 z = 16384，即为浮点 $z = 16384/2^{15} = 0.5$。

3.3.4 程序变量的 Q 值确定

在前面几节介绍的例子中，由于 x、y、z 的值都是已知的，因此从浮点变为定点时 Q 值容易确定。在实际的 DSP 应用中，程序中参与运算的有常数，更多的是变量。那么如何确定程序中变量的 Q 值呢？

从前面的分析可以知道，确定变量的 Q 值实际上就是确定变量的动态范围，动态范围确定了，则 Q 值也就确定了。

设变量的绝对值的最大值为 $|\max|$，注意 $|\max|$ 必须小于或等于 32767。取一个整数 n，使满足：

$$2^{n-1} < |\max| < 2^n$$

则有：

$$2^{-Q} = 2^{-15} \times 2^n = 2^{-(15-n)}$$

$Q = 15-n$

例如，某变量的值在−1 至＋1 之间，即 $|\max|<1$，因此 $n = 0$，$Q = 15-n = 15$。

既然确定了变量的$|\max|$就可以确定其 Q 值，那么变量的$|\max|$又是如何确定的呢？一般来说，确定变量的$|\max|$有两种方法，一种是理论分析法，另一种是统计分析法。

1．理论分析法

有些变量的动态范围通过理论分析是可以确定的。例如：

（1）三角函数，$y=\sin（x）$或$y=\cos（x）$，由三角函数知识可知，$|y|\leqslant 1$。

（2）汉明窗，$y（n）=0.54-0.46\cos[2\pi n/（N-1）]$，$0\leqslant n\leqslant N\text{-}1$。因为$-1\leqslant\cos[2\pi n/（N-1）]\leqslant 1$，所以 $0.08\leqslant y（n）\leqslant 1.0$。

（3）FIR 卷积。$Y（n）=\sum_{k=0}^{N-1}h(k)x(n-k)$，设$\sum_{k=0}^{N-1}|h(k)|=1.0$，且 $x（n）$是模拟信号 12 位量化值，即有 $|x(n)|\leqslant 2^{11}$，则$|y(n)|\leqslant 2^{11}$。

（4）理论已经证明，在自相关线性预测编码（LPC）中，反射系数 k_i 满足下列不等式：$|k_i|<1.0$，$i=1,2,\cdots,p$，p 为 LPC 的阶数。

2．统计分析法

对于理论上无法确定范围的变量，一般采用统计分析的方法来确定其动态范围。所谓统计分析，就是用足够多的输入信号样值来确定程序中变量的动态范围，这里输入信号一方面要有一定的数量，另一方面必须尽可能地涉及各种情况。例如，在语音信号分析中，统计分析时就必须采集足够多的语音信号样值，并且在所采集的语音样值中，应尽可能地包含各种情况，如音量的大小，声音的种类（男声、女声等）。只有这样，统计出来的结果才能具有典型性。

当然，统计分析毕竟不可能涉及所有可能发生的情况，因此，对统计得出的结果在程序设计时可采取一些保护措施，如适当牺牲一些精度，Q 值取比统计值稍大些，使用 DSP 芯片提供的溢出保护功能等。



3.3.5　浮点至定点变换的 C 程序举例

本节通过一个例子来说明 C 程序从浮点变换至定点的方法。这是一个对语音信号（0.3~3.4 kHz）进行低通滤波的 C 语言程序，低通滤波的截止频率为 800 Hz，滤波器采用 19 点有限冲激响应 FIR 滤波。语音信号的采样频率为 8 kHz，每个语音样值按 16 位整型数存放在 insp.dat 文件中，滤波后的数存放在 outsp.dat 中。

【例 3.9】语音信号 19 点 FIR 800 Hz 低通滤波 C 语言浮点程序

```
#include <stdio.h>
#define   length    180              /*语音帧长为 180 点 = 22.5ms@8 kHz采样*/
void      filter（int xin[],int xout[],int n,float h[]）;        /*滤波子程序说明*/
/*19 点滤波器系数*/
static    float h[19]=
            {0.01218354,-0.009012882,-0.02881839,-0.04743239,-0.04584568,
             -0.008692503,0.06446265,0.1544655,0.2289794,0.257883,
             0.2289794,0.1544655,0.06446265,-0.008692503,-0.04584568,
             -0.04743239,-0.02881839,-0.009012882,0.01218354};
static    int x1[length+20];
```

```
/*低通滤波浮点子程序*/
void      filter（int xin[],int xout[],int n,float h[]）
          {
               int i,j;
               float sum;
               for（i=0;i<length;i++）  x1[n+i-1]=xin[i];
               for  （i=0;i<length;i++）
                    {
                      sum=0.0;
                      for（j=0;j<n;j++）  sum+=h[j]*x1[i-j+n-1];
                      xout[i]=（int）sum;
                    }
               for（i=0;i<（n-1）;i++）  x1[n-i-2]=xin[length-1-i];
          }

/*主程序*/
void   main（）
{
    FILE      *fp1,*fp2;
     int   i;
    int          frame,indata[length],outdata[length];
    fp1=fopen（“insp.dat”,"rb"）;                    /*输入语音文件*/
    fp2=fopen（“outsp.dat”,"wb"）;                   /*滤波后语音文件*/

    frame=0;
    while（feof（fp1）==0）
     {
       frame++;
       printf（"frame=%d\n",frame）;
       for（i=0;i<length;i++）    indata[i]=getw（fp1）; /*取一帧语音数据*/
       filter（indata,outdata,19,h）                    ; /*调用低通滤波子程序*/
       for（i=0;i<length;i++）    putw（outdata[i],fp2）; /*将滤波后的样值写入文件*/
      }
     fcloseall（）                                      ; /*关闭文件*/
     return（0）;
}
```

【例 3.10】语音信号 19 点 FIR 800 Hz 低通滤波 C 语言定点程序

```
#include <stdio.h>
const int length=180;
void   filter（int xin[],int xout[],int n,int h[]）;
static int   h[19]={399,-296,-945,-1555,-1503,-285,2112,5061,7503,8450,
                7503,5061,2112,-285,-1503,-1555,-945,-296,399};      /*Q15*/
static int   x1[length+20];
```

```
/*低通滤波定点子程序*/
void   filter（int xin[],int xout[],int n,int h[]）
        {
            int i,j;
            long sum;
            for（i=0;i<length;i++）  x1[n+i-1]=xin[i];
            for  （i=0;i<length;i++）
                {
                  sum=0;
                  for（j=0;j<n;j++）    sum+=（long）h[j]*x1[i-j+n-1];
                  xout[i]=sum>>15;
                 }
            for（i=0;i<（n-1）;i++）  x1[n-i-2]=xin[length-i-1];
        }
```

主程序与浮点程序中主程序完全一样。

3.4　DSP 定点算术运算实现的基本原理

定点 DSP 芯片的数值表示基于 2 的补码表示形式。每个 16 位数用 1 个符号位、i 个整数位和 $15-i$ 个小数位来表示。因此，二进制数 00000010.10100000 表示的值为 $2^1+2^{-1}+2^{-3}=2.625$，这个数可用 Q8 格式（8 个小数位）来表示，其表示的数值范围为−128 至+127.996，一个 Q8 定点数的小数精度为 $1/2^8$=0.004。

虽然特殊情况（如动态范围和精度要求）必须使用混合表示法。但是，更通常的是全部以 Q15 格式表示的小数或以 Q0 格式表示的整数来工作。这一点对于主要是乘法和累加的信号处理算法特别现实，小数乘以小数得小数，整数乘以整数得整数。当然，乘积累加时可能会出现溢出现象，在这种情况下，程序员应当了解实际的物理过程以注意可能的溢出情况。下面我们来讨论乘法、加法和除法的 DSP 定点运算，汇编程序以 TMS320C54x 为例。

3.4.1　定点乘法

两个定点数相乘的规则是：如果两个被乘数的格式分别是 Qx 和 Qy，则乘法的结果将为 Q（$x+y$） 格式，即：Qx×Qy=Q（x+y）。

下面分三种情况进行讨论。

1．小数乘小数

Q15×Q15 = Q30

【例 3.11】 0.5×0.5 = 0.25

```
    0.100000000000000      ;Q15
×   0.100000000000000      ;Q15
-------------------------------
  00.010000000000000000000000000000=0.25  ;Q30
```

两个 Q15 的小数相乘后得到一个 Q30 的小数，即有两个符号位。一般情况下相乘后得到

的满精度数不必全部保留，而只需保留 16 位单精度数。由于相乘后得到的高 16 位不满 15 位的小数精度（Q14），为了达到 15 位精度，须将乘积左移一位，得到 Q15 精度。

下面是上述乘法的 TMS320C54x 程序：

```
SSBX    FRCT        ; FRCT=1，乘积自动左移 1 位
LD      OP1,16,A    ; OP1=4000H（0.5/Q15），放入累加器 A 高 16 位
MPYA    OP2         ; OP2=4000H（0.5/Q15），ACC 的高 16 位与 OP2 相乘
                    ; ACC = 2000H（0.25/Q15），乘积结果放入 ACC
```

2．整数乘整数

Q0×Q0 = Q0

【例 3.12】17×（–5）= –85

```
  0000000000010001=17
×  1111111111111011= –5
-----------------------------------
  11111111111111111111111110101011= –85
```

3．混合表示法

许多情况下，运算过程中为了既满足数值的动态范围又保证一定的精度，就必须采用 Q0 与 Q15 之间的表示法。比如，数值 1.2345，显然 Q15 无法表示，而若用 Q0 表示，则最接近的数是 1，精度无法保证。因此，数 1.2345 最佳的表示法是 Q14。

【例 3.13】1.5×0.75 = 1.125

```
  01.10000000000000 = 1.5          ;Q14
×  00.11000000000000 = 0.75         ;Q14
-----------------------------------------------
  0001.00100000000000000000000000000 = 1.125     ;Q28
```

Q14 的最大值不大于 2，因此，两个 Q14 数相乘得到的乘积不大于 4。

一般地，若一个数的整数位为 i 位，小数位为 j 位，另一个数的整数位为 m 位，小数位为 n 位，则这两个数的乘积为 （$i+m$ ）位整数位和（$j+n$）位小数位。这个乘积的最高 16 位可能的精度为（$i+m$ ）整数位和（$15-i-m$）小数位。

但是，若事先了解数的动态范围，就可以增加数的精度。例如，程序员了解到上述乘积不会大于 1.8，就可以用 Q14 数表示乘积，而不是理论上的最佳情况 Q13。例 3.13 的 TMS320C54x 程序如下。

```
SSBX    FRCT        ; FRCT=1,乘法结果左移 1 位
LD      OP1,16,A    ; OP1 = 6000H（1.5/Q14）
MPYA    OP2         ; OP2 = 3000H（0.75/Q14）
                    ; ACC = 2400H（1.125/Q13）
```

上述方法，为了精度均对乘的结果舍位，结果所产生的误差相当于减去一个 LSB（最低位）。采用下面简单的舍入方法，可使误差减少二分之一。

```
SSBX    FRCT        ; FRCT=1，乘法结果左移 1 位
LD      OP1, T      ; OP1 放入 T 寄存器
MPYR    OP2, A      ; OP2*T+2^15，并将低 16 位清零
                    ; A（32-16）位为乘运算并舍入后的结果
SACH    ANS
```

上述程序说明，不管 ANS 为正或负，所产生的误差是 1/2 LSB。

3.4.2　定点加法

乘的过程中，程序员可不考虑溢出而只需调整运算中的小数点。而加法则是一个更加复杂的过程。首先，加法运算必须用相同的 Q 点表示；其次，程序员或者允许其结果有足够的高位以适应位的增长，或者必须准备解决溢出问题。如果操作数仅为 16 位长，其结果可用双精度数表示。下例为加法运算并保留 32 位结果至累加器 A。

```
LD      OP1, A   ;（Q15）
ADD     OP2, A   ;（Q15）
```

加法运算最可能出现的问题是运算结果溢出。DSP 提供了溢出标志 OVA 和 OVB，此外，使用溢出保护功能（OVM = 1）可使累加结果溢出时累加器饱和为绝对值最大的正数或负数。当然，即使如此，运算精度还是大大降低。因此，最好的方法是完全理解基本的物理过程并注意选择数的表达方式。

3.4.3　定点除法

在通用 DSP 芯片中，一般不提供单周期的除法指令，为此必须采用除法子程序来实现。二进制除法是乘法的逆运算。乘法包括一系列的移位和加法，而除法可分解为一系列的减法和移位。下面我们来说明除法的实现过程。

假设累加器的长度是 8 位，做的除法运算为 10 除以 3。除的过程就是除数逐步移位并与被除数比较的过程，在每一步进行减法运算，如果能减则将位插入商中。下面给出了除的过程。

（1）除数的最低有效位对齐被除数的最高有效位。

```
    00001010
－   00011000
    11110010
```

（2）由于减法结果为负，放弃减法结果，将被除数左移一位，再减。

```
    00010100
－   00011000
    11111000
```

（3）结果仍为负，放弃减法结果，被除数左移一位，再减。

```
    00101000
－   00011000
    00010000
```

（4）结果为正，将减法结果左移一位后加 1 ，作最后一次减。

```
    00100001
－   00011000
    00001001
```

（5）结果为正，将结果左移一位加 1 得最后结果。高 4 位代表余数，低 4 位表示商。

00010011

即商为 0011=3，余数为 0001=1。

一般的 DSP 都没有专门的除法指令，但使用 DSP 指令集中的条件减指令 SUBC 可以完成有效灵活的除法功能。使用这一指令的唯一限制是两个操作数必须为正。程序员事先要了解其可能的运算数的特性，如其商是否可以用小数表示及商的精度是否可以计算出来。

实际上，对于求 *x*/*y*，我们也可以将它看做 *x* 与 1/*y* 相乘。因此，先求得 *y* 的倒数，再将 *x* 与它相乘，就可以得到 *x*/*y* 的值了。

定点除法可以通过重复使用 DSP 芯片指令集中所提供的条件减指令 SUBC 来实现。给定一个 16 位的正除数和正被除数，重复 SUBC 指令 16 次，将在累加器的低 16 位产生 16 位的商，在累加器的高 16 位产生 16 位的余数。

对于一个数 *y*，采用 SUBC 求取其倒数 1/*y* 的过程如下：

（1）设除数 *y* 用 Q*n* 来表示，则该除数可以表示为：$y \times 2^n$；

（2）对 $y \times 2^n$ 取绝对值；

（3）将被除数 1 用 Q15 表示，并存放在累加器的低 16 位（ACCL）：$\mathrm{ACCL} = 1 \times 2^{15}$；

（4）重复 SUBC 指令 16 次，在 ACCL 中得到以 Q（15−*n*）格式表示的商：

$$\frac{1 \times 2^{15}}{y \times 2^n} = \frac{1}{y} \times 2^{(15-n)}$$；

（5）如果输入为负数，则将商变为负数。

下面，给出一个求 1/*y* 的 TMS320C54/55x 的 16 位程序。如果程序中输入 *y* 用 *Qn*（*n*<16）表示的话，则输出 1/*y* 用 *Q*（15−*n*）来表示。

【例 3.14】16 位定点数求倒数的 DSP 程序

```
; 外部符号定义
                .def    inv2_input, inv2_output
; 局部符号定义
inv2_input      .usect "qmath",1
inv2_output     .usect "qmath",1
temp            .usect "qmath",1

qinv2:          SETC    OVM             ; 溢出保护
                SETC    SXM             ; 符号扩展
                SPM     #0
                LAR     AR2,#inv2_input ; AR2 指向 inv2_input
                LAR     AR0,#temp       ; AR0 指向 temp
                MAR     *,AR2           ; 默认寄存器为 AR2
;================================================================
                LACC    *,16,AR0        ; 将输入装载到 ACC 的高 16 位
                                        ; 默认寄存器为 AR0
                BCND    calculate,NEQ   ; 如果输入为 0，则返回结果为 0
                LACC    #0000h          ; ACC=0
                B       return
```

```
calculate:    BCND    positive, GT        ; 输入为正数，跳转到 positive
              ABS                         ; 输入为负数，取绝对值
positive:     SACH    *                   ; 存储取绝对值后的输入
              LACC    #01h,15             ; 将 Q15 格式的数 1 装载到 ACC 的低 16 位
              RPT     #15                 ; 重复下一条指令 16 次
              SUBC    *                   ; 条件减
              AND     #0ffffh
              SACL    *
              BIT     *,0,AR2
              BCND    isnot1,NTC
              LACC    #7fffh
isnot1:       BIT     *,0
              BCND    return,NTC
              NEG
return:       CLRC    OVM
;================================================================
              MAR     *,AR2
              LAR     AR2,#inv2_output
              SACL    *
              RET
```

上述程序可以作为子程序调用。其中，输入存放在 inv2_input 中，输出存放在 inv2_output 中。例如，设输入 y=8.125，用 Q8 表示为 $8.125 \times 2^8 = 2080$，即 0820h。输出是一个用 Q（15−8）=Q7 表示的数，8.125 的倒数是 0.1230769，用 Q7 表示为 15，即十六进制的 0 Fh。

3.5　非线性运算定点实现方法

在数值运算中，除基本的加减乘除运算外，还有其他许多非线性运算，如，三角函数运算、对数运算、开方运算、指数运算等。实现这些非线性运算通常可以采用以下三种方法。

3.5.1　级数展开法

实现非线性运算最方便的方法是直接调用 DSP 编译系统所提供的库函数，TMS320 定点 DSP 芯片的 C 编译器提供了比较丰富的运行支持库函数。在这些库函数中，包含了诸如三角函数、对数、开方、指数等常用的非线性函数。在 C 程序中（也可在汇编程序中）只要采用与库函数相同的变量定义，就可以直接调用。

DSP 库函数中实现非线性运算一般采用级数展开法。例如，正弦函数和对数函数可以分别表示为如下的级数展开：

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \cdots, |x| < 1$$

直接调用库函数的优点是编程方便、运算精度高，但缺点是运算量和程序量都比较大，在

实时 DSP 中应用受到一定的限制。

下面两节分别介绍两种常用的快速实现方法：查表法和混合法。

3.5.2 查表法

在实时 DSP 应用中实现非线性运算，一般会适当降低运算精度来提高程序的运算速度。查表法是快速实现非线性运算最常用的方法。采用这种方法，必须根据自变量的范围和精度要求制作一张表格。显然输入的范围越大，精度要求越高，则所需的表格就越大，即存储量也越大。查表法求值所需的计算就是根据输入值确定表的地址，根据地址就可得到相应的值，因而运算量较小。查表法比较适合于非线性函数是周期函数或已知非线性函数输入值范围这两种情况，例 3.15 和例 3.16 分别说明这两种情况。

【例 3.15】已知正弦函数 y=sin（x），制作一个 512 点表格，并说明查表方法。

由于正弦函数是周期函数，函数值在－1 至＋1 之间，用查表法比较合适。

由于 Q15 的表示范围为－1 至 32767/32768 之间，原则上讲－1 至＋1 的范围必须用 Q14 表示。但一般从方便和总体精度考虑，类似情况仍用 Q15 表示，此时＋1 用 32767 来表示。

（1）产生 512 点值的 C 语言程序如下所示:

```
#define   N   512
#define   pi   3.14159
int    sin_tab[512];
void   main（）
{
      int i;
      for（i=0; i <N;i++）   sin_tab[i]=（int）（32767*sin（2*pi*i/N））;
}
```

（2）查表

查表实际上就是根据输入值确定表的地址。设输入 x 在 0~2π之间，则 x 对应于 512 点表的地址为：

```
index =  （int）（512*x/2π）
则  y = sin（x）  = sin_tab[index]
```

如果 x 用 Q12 定点数表示，将 512/2π用 Q8 表示为 20861，则计算正弦表的地址的公式为：

```
index =（x*20861L）>>20
```

【例 3.16】用查表法求以 2 为底的对数，已知自变量值范围为 0.5~1，要求将自变量范围均匀划分为 10 等分。试制作这个表格并说明查表方法。

（1）制表。y=$\log_2$（x），由于 x 在 0.5 到 1 之间，因此 y 在−1 到 0 之间，x 和 y 均可用 Q15 表示。由于对 x 均匀划分为 10 段，因此，10 段对应于输入 x 的范围如表 3-2 所示。若每一段的对数值都取第一点的对数值，则表中第一段的对数值为 y_0（Q15）=（int）（$\log_2$（0.5）×32768），第二段的对数值为 y_1（Q15）=（int）（$\log_2$（0.55）×32768），依次类推，如表 3-2 所示。

（2）查表。查表时，先根据输入值计算表的地址，计算方法为：

```
index =  （（x−16384）×20）>>15;
```

式中，index 就是查表用的地址。例如，已知输入 x = 26869，则 index = 6，因此，y = –10549。

表 3-2　logtab0　10 点对数表（输入 0.5~1）

地址	输入值	对数值（Q15）
0	0.50~0.55	-32768
1	0.55~0.60	-28262
2	0.60~0.65	-24149
3	0.65~0.70	-20365
4	0.70~0.75	-16862
5	0.75~0.80	-13600
6	0.80~0.85	-10549
7	0.85~0.90	-7683
8	0.90~0.95	-4981
9	0.95~1.00	-2425

3.5.3　混合法

1．提高查表法的精度

上述方法查表所得结果的精度随表的大小而变化，表越大，则精度越高，但存储量也越大。当系统的存储量有限而精度要求也较高时，查表法就不太适合。那么能否在适当增加运算量的情况下提高非线性运算的精度呢？

下面介绍一种查表结合少量运算来计算非线性函数的混合法，这种方法适用于在输入变量的范围内函数呈单调变化的情形。

混合法是在查表的基础上采用计算的方法以提高当输入值处于表格两点之间时的精度。提高精度的一个简便方法是采用折线近似法，如图 3-1 所示。

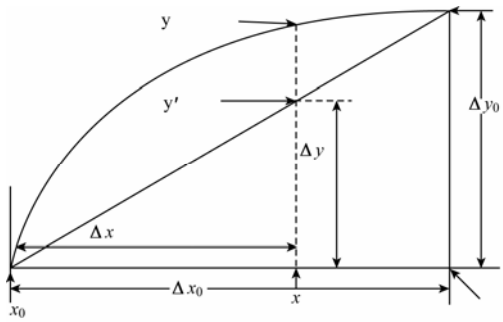


图 3-1　提高精度的折线近似法

仍以求以 2 为底的对数为例（例 3.16）。设输入值为 x，则精确的对数值为 y，在表格值的两点之间作一直线，用 y' 作为 y 的近似值，则有：

$$y' = y_0 + \Delta y$$

其中 y_0 由查表求得。现在只需在查表求得 y_0 的基础上增加 Δy 即可。Δy 的计算方法如下：

$$\Delta y = (\Delta x / \Delta x_0)\ \Delta y_0 = \Delta x\ (\Delta y_0 / \Delta x_0)$$

式中，$\Delta y_0 / \Delta x_0$ 对每一段来说是一个恒定值，可作一个表格直接查得。此外计算 Δx 时需用到每段横坐标的起始值，这个值也可作一个表格。这样共有三个大小均为 10 的表格，分别为存储每段起点对数值的表 logtab0、存储每段 $\Delta y_0 / \Delta x_0$ 值的表 logtab1 和存储每段输入起始值 $x0$ 的表 logtab2，表 logtab1 和表 logtab2 可用下列两个数组表示。

```
int     logtab1[10]={22529,20567,18920,17517,16308,
                     15255,14330,13511,12780,12124};    /*Δy0/Δx0 : Q13*/
int     logtab2[10]={16384,18022,19660,21299,22938,
                     24576,26214,27853,29491,31130};    /* x0: Q15*/
```

综上所述，采用混合法计算对数值的方法可归纳为：

（1）根据输入值，计算查表地址：index=（（x-16384）×20）>>15;

（2）查表得 y_0=logtab0[index];

（3）计算Δx=x−logtab2[index];

（4）计算Δy=（Δx×logtab1[index]）>>13;

（5）计算得结果 y=y_0+Δy。

【例 3.17】已知 x=0.54，求 $\log_2$（x）。

0.54 的精确对数值为 y=$\log_2$（0.54）=−0.889。

混合法求对数值的过程为：

（1）定标 Q15，定标值 x=0.54*32768=17694；

（2）表地址 index=（（x−16384）×20）>>15=0;

（3）查表得 y_0=logtab0[0]=-32768;

（4）计算Δx=x−logtab2[0]=17694−16384=1310;

（5）计算Δy=（Δxlogtab1[0]）>>13=（1310*22529L）>>13=3602;

（6）计算结果 y=y0+Δy=−32768+3602=−29166。

结果 y 为 Q15 定标，折算成浮点数为−29166/32768=−0.89，可见精度较高。

2．扩大自变量范围

如上所述，查表法比较适用于周期函数或自变量的动态范围不是太大的情形。对于像对数这样的非线性函数，输入值和函数值的变化范围都很大。如果输入值的变化范围很大，则作表就比较困难。那么能否比较好地解决这个问题，既不使表格太大，又能得到比较高的精度呢？

下面来讨论一种切实可行的方法。

设 x 是一个大于 0.5 的数，则 x 可以表示为下列形式：

$$x = m \cdot 2^e$$

式中，0.5≤ m≤1.0，e 为整数。则求 x 的对数可以表示为：

$\log_2$（x） = $\log_2$（$m \cdot 2^e$） = $\log_2$（m） + $\log_2$（2^e） = e + $\log_2$（m）

也就是说，求 x 的对数实际上只要求 m 的对数就可以了，而由于 m 的数值在 0.5 和 1.0 之间，用上面介绍的方法是完全可以实现的。例如：

$\log_2$（10000）= $\log_2$（0.61035× 2^{14}）= $\log_2$（0.61035）+ 14 = 13.2877

可见，如果一个数可以用比较简便的方法表示为上面的形式，则求任意大小数的对数也比较方便的。TMS320 定点 DSP 芯片的指令集中提供了用于对 ACC 中的数进行规格化的指令 NORM 和 EXP，其作用是使累加器中的数左移，直至数的最高位被移至累加器的第 30 位。

对一个 16 位整数 x 进行规格化处理实际上就是做这样一个等效变换：

$$x = \frac{x \cdot 2^Q}{32768} \cdot 2^{15-Q}$$

【例 3.18】实现以 2 为底的对数的 C 定点模拟程序

```
int     logtab0[10] = {−32768,−28262,−24149,  −20365,−16862,
                       −13600,−10549,−7683,−4981,−2425};   /*Q15*/
int     logtab1[10] = {22529,20567,18920,17517,16308,
                       15255,14330,13511,12780,12124};     /*Q13*/
int     logtab2[10] = {16384,18022,19660,21299,22938,
```

```
                    24576,26214,27853,29491,31130};    /*Q15*/
int     log2_fast（int   Am）
{
        int     point,point1;
        int     index,x0,dx,dy,y;
        point = 0;
        while（Am<16384）  {point++；Am = Am<<1;}     /*对 Am 进行规格化*/
        point1 =  （15－point－4）*512;               /*输入为 Q4，输出为 Q9*/
        index =  （（Am−16384）*20L）>>15;            /*求查表地址*/
        dx = Am-logtab2[index];
        dy =  （（long）dx*logtab1[index]）>>13;
        y =  （dy+logtab0[index]）>>6;                /*Q9*/
        y = point1+y;
        return  （y）;
}
```

上述程序中，输入值 Am 采用 Q4 表示，输出采用 Q9 表示，如果输入输出的 Q 值与上面程序中的不同，则应作相应的修改。

3.6　浮点数的表示格式

一个浮点数通常采用指数和尾数的表示形式：

$$x = m \times 2^e$$

式中，e 称为指数，m 称为尾数。尾数通常用归一化数表示，可以分为符号（s）和分数（f）两部分。

在二进制表示中，符号用 1 位表示。由于表示尾数和分数的位数不同可以分为不同精度的浮点数格式，如 IEEE 定义的单精度浮点格式、双精度浮点格式等。即使是同一种精度的浮点格式，由于尾数域、符号域和指数域的位置不同，表示的浮点格式也不相同，如 IEEE 单精度浮点格式和 TMS320C3x 单精度浮点格式是不一样的。下面我们以常用的单精度浮点格式为例介绍这两种浮点格式。

3.6.1　IEEE 浮点数格式

IEEE 754-1985 标准定义了四种浮点数的格式，即单精度格式、扩展单精度格式、双精度格式和扩展双精度格式。由于在 DSP 芯片中，单精度格式用得最多，因此，下面介绍 IEEE 单精度浮点格式的表示方法。

IEEE 754-1985 定义的单精度浮点格式如图 3-2 所示。

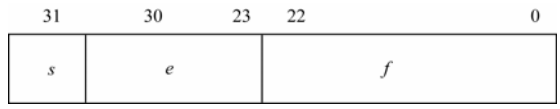


图 3-2　IEEE 单精度浮点格式

在这个格式中，浮点数的总长度为 32 位，其中，

s 是尾数 m 的符号位，共 1 位。$s = 0$ 表示正数，$s = 1$ 表示负数。

e 是指数，用无符号数表示，共 8 位，取值范围为 0~255。

f 是尾数的分数部分，共 23 位。

一个 IEEE 单精度浮点数 x 用 e、s、f 表示可分为以下 5 种情况：

（1）如果 0<e<255，则 $x=(-1)^s 2^{e-127}(1.f)$，式中 $s=0$ 或 1，而 e 和 f 均用十进制表示；

（2）如果 $e=0$ 且 $f\neq 0$，则 x 是一个非归一化的数，且 $x=(-1)^s 2^{-126}(0.f)$，$s=0$ 或 1，f 用十进制表示；

（3）如果 $e=0$ 且 $f=0$，则 $x=0$;

（4）如果 $e=255$ 且 $f\neq 0$，则 x 是一个无效数（NaN，Not a Number）；

（5）如果 $e=255$ 且 $f=0$，则 x 为无穷大。

从第一和第二两种情况可以看出，由于小数点左边的 1 或 0 是隐含的，因此实际上 IEEE 单精度格式浮点数的尾数精度为 24 位。

例：

（1）一个 32 位浮点数为：1,11000010,10101100000000000000000

符号位：　=1，表示是一个负数

尾数：　$=1+2^{-1}+2^{-3}+2^{-5}+2^{-6}=1.6875$

指数：　$=2^7+2^6+2^1=194$

十进制等值数：$=2^{-3}+2^{-5}+2^{-6}=-1.6875\times 2^{(194-127)}=-2.49031044995\times 10^{20}$

（2）一个 32 位浮点数为：0,01100010,01111100000000100000000

符号位：　=0，表示是一个正数

尾数：　$=1+2^{-2}+2^{-3}+2^{-4}+2^{-5}+2^{-6}+2^{-15}=1.48440551758$

指数：　$=2^6+2^5+2^1=98$

十进制等值数：$=1.48440551758\times 2^{(98-127)}=2.76492073681\times 10^{-9}$

3.6.2　TMS320C3x 浮点数格式

为了在硬件实现时获得更高的效率，TMS320C3x 采用了不同于 IEEE 标准的浮点格式。TMS320C3x 共定义了三种浮点格式。与 IEEE 浮点格式一样，每种浮点格式都可以分为三个区域，即指数域（e）、1 位符号域（s）和分数域（f）。图 3-3 示出了 TMS320C3x 单精度浮点数格式，从图中可以看出 TMS320C3x 与 IEEE 浮点格式的区别。

其中，指数域是一个二进制补码数，而符号域和分数域合起来称为尾数（m）。尾数是一个归一化的二进制补码数，其绝对值在 1 到 2 之间。一个 TMS320C3x 的浮点数 x 用指数 e、符号 s 和分数 f 可以表示为：

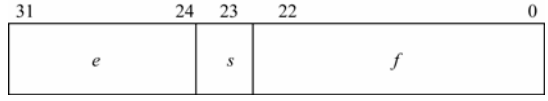


图 3-3　TMS320C3x 单精度浮点数格式

$$x=\{(-2)^s+(.f)\}2^e$$

即

$$m=\{(-2)^s+(.f)\}$$

另一种描述尾数的方法是 $m=ssf$，则浮点数 x 的表示可以进一步分为以下三种情况：

（1）如果 $s=0$，则 $x=01.f\times 2^e$；

（2）如果 $s=1$，则 $x=10.f\times 2^e$；

（3）如果 e 等于绝对值最大的负数，且 $s=f=0$，则 $x=0$。

在 TMS320C3x 的三种浮点格式中，第一种称为短浮点格式，用作立即操作数，这种格式包括 4 位指数，1 位符号和 11 位分数，尾数共 12 位，总长度为 16 位。当 $e=-8$，$s=f=0$ 时表示的数为 0。

第二种称为单精度格式，包括 8 位指数，1 位符号和 23 位分数，尾数共 24 位，总长度为 32 位。当 $e=-128$，$s=f=0$ 时表示的数为 0。这种格式是最常用的格式，在 TMS320C3x 中，大部分操作都是基于这种浮点格式。例如一个浮点数 x，十六进制表示为 217FFFFFh，用 $m\times 2^e$ 表示为：

$$2^{33}\times 01.111\ 1111\ 1111\ 1111\ 1111\text{b}$$

则十进制为 1.71798682×10^{10}。

第三种称为扩展精度格式，包括 8 位指数、1 位符号和 31 位分数，尾数共 32 位，总长度为 40 位，$e=-128$，$s=f=0$ 时表示的数为 0。

由于在 TMS320C3x 中用的最多的是单精度格式，下面以这种格式为例说明浮点数的表示方法和表示范围。

1．表示方法

（1）设浮点数 x 的 32 位十六进制用 F 表示，且 $F=$0x00000000，则指数 $e=00$（第 24~31 位），符号 $s=0$（第 23 位），分数 $f=0$（第 0~22 位），因为 $s=0$，故浮点数 x 的二进制值为 $01.f\times 2^e=01.0\times 2^0=1$，十进制值也为 1。

（2）设 $F=$0x02400000，则 $e=$0x02 = 2，s = 0，$f=100\ 0000\ 0000\ 0000\ 0000\ 0000$，浮点数 x 的二进制值为 $01.10\times 2^2=0110$B，十进制值为 6，式中舍去了 f 尾部的 0，b 表示二进制。

（3）设 $F=$0x03A00000，则 $e=$0x03 = 3，$s=1$，$f=010$（已截尾），浮点数 x 的二进制值为 $10.010\times 2^3=10010$B，因为 $s=1$，为负数，所以十进制值为－[01101b+1] = －14。

（4）设 $F=$0xFC20 0000，指数 $e=$0xFC，$s=0$，$f=010$（已截尾），因为 e=0xFC，其十进制数值为 －[0000 0011b+1]=－4，所以浮点数 x 的十进制值为：

$$01.010\times 2^{-4}=0.0001010\text{B}=2^{-4}+2^{-6}=\frac{5}{64}$$

2．表示范围

（1）当 $s=0$，$e=127$，f 的 23 位均为 1 时，表示的浮点数为最大的正数：

$$m=01.1111\ 1111\ 1111\ 1111\ 1111\ 111=2-2^{-23}$$

$$x=(2-2^{-23})\times 2^{127}=3.4028234\times 10^{38}$$

（2）当 $s=0$，$e=-127$，f 的 23 位均为 0 时，表示的浮点数为最小的正数：

$$m=01.0000\ 0000\ 0000\ 0000\ 0000\ 000=1$$

$$x=1\times 2^{-127}=5.8774717\times 10^{-39}$$

（3）当 $s=1$，$e=-127$，f 的 23 位均为 1 时，表示的浮点数为绝对值最小的负数：

$$m=10.1111\ 1111\ 1111\ 1111\ 1111\ 111=-1-2^{-23}$$

$$x=(-1-2^{-23})\times 2^{-127}=-5.8774724\times 10^{-39}$$

（4）当 $s=1$，$e=127$，f 的 23 位均为 0 时，表示的浮点数为绝对值最大的负数：

$$m=10.0000\ 0000\ 0000\ 0000\ 0000\ 000=-2$$

$$x=-2\times 2^{127}=-3.4028236\times 10^{38}$$

在 TMS320C3x 浮点运算过程中，上述三种浮点格式可以自动转换。如，短浮点格式转换为

单精度格式，扩展精度浮点格式转换为单精度格式等。转换由硬件自动实现，不增加额外的运算。

3.7 基本的浮点运算

3.7.1 浮点乘法和加减法

浮点 DSP 芯片一般都提供单周期的浮点乘法和加减法。下面简单介绍一下浮点乘法和加减法的操作过程。

设两个浮点数 $x1$ 和 $x2$ 分别表示为：

$$x1 = m1 \times 2^{e1}$$

$$x2 = m2 \times 2^{e2}$$

式中，$m1$ 和 $m2$ 分别是 $x1$ 和 $x2$ 的尾数，$e1$ 和 $e2$ 分别是 $x1$ 和 $x2$ 的指数。

$x1$ 和 $x2$ 相乘包括以下几个过程：（1）尾数相乘，（2）指数相加，（3）对得到的乘积进行归一化和特殊情况处理。

浮点数加法和减法的原理是一样的。与乘法相比，加减法的过程稍微复杂些。首先要对指数小的数按指数大的数进行归正。设 $e1>e2$，则需对 $x2$ 进行如下归正：

$$x2 = m2 \times 2^{-(e1-e2)} 2^{e1}$$

然后进行如下运算：

$$x3 = [m1 \pm m2 \times 2^{-(e1-e2)}]2^{e1}$$

如果 $e1<e2$，则需对 $x1$ 进行归正，其方法是完全一样的。

3.7.2 浮点除法

浮点 DSP 芯片一般不提供专门的除法指令，因此实现浮点除法必须用子程序来实现。实现浮点除法可以采用近似迭代的方法来实现。下面介绍一种用迭代法先实现浮点数的倒数，然后用被除数乘以除数的倒数来实现浮点除法的方法。

设 $z = x/y$，如果先求得 y 的倒数 $1/y$，则将 x 乘以 y 的倒数即可求得 x/y。求 y 的倒数的迭代方法为：

$$e[i] = e[i-1] \times （2.0 - y\, e[i-1]）$$

采用这个迭代公式需要一个初始估值 $e[0]$，设 $y = m \times 2^{e}$，则 $e[0]$的一个较好估值为：

$$e[0] = 1.0 \times 2^{-e-1}$$

例 3.19 是求浮点数倒数的 TMS320C3x 的汇编程序，输入数 v 和其倒数均在寄存器 R0 中，这个程序采用 5 次迭代。要想获得更高的精度，增加迭代次数即可。

【例 3.19】浮点数求倒数的 TMS320C3x 汇编程序

```
INVF:
LDF     R0,R3     ; 保存输入数
ABSF    R0        ; 求绝对值
PUSHF   R0
POP     R1
ASH     -24,R1    ; R1 的低 8 位包含指数部分
NEGI    R1
```

```
SUBI    1,R1
ASH     24,R1
PUSH    R1
POPF    R1          ; R1 = i[0] = 1.0*2**（-e-1）
MPYF    R1,R0,R2    ; 第一次迭代
SUBRF   2.0,R2
MPYF    R2,R1
MPYF    R1,R0,R2    ; 第二次迭代
SUBRF   2.0,R2
MPYF    R2,R1
MPYF    R1,R0,R2    ; 第三次迭代
SUBRF   2.0,R2
MPYF    R2,R1
MPYF    R1,R0,R2    ; 第四次迭代
SUBRF   2.0,R2
MPYF    R2,R1
RND     R1          ; 最后一次迭代采用 e[5]=（e[4]*（1.0－（v*e[4]））+[4]
 MPYF   R1,R0,R2
SUBRF   1.0,R2
MPYF    R1,R2
ADDF    R2,R1
RND     R1,R0       ; 处理负数情况
NEGF    R0,R2
LDF     R3,R3
LDFN    R2,R0       ; if  v < 0,R0 =  －R0
RETS
```

3.8　非线性浮点运算的快速实现

在实际的 DSP 应用中，经常需要实现非线性的运算，如对数、指数、三角函数等。由于 TMS320 浮点 C 编译器提供了比较丰富的库函数，因此实现这些非线性运算最方便的方法当然是直接调用库函数。如果程序是用 C 语言编写，则这个工作直接由编译器和链接器完成；如果程序用汇编语言编写，则在汇编程序中调用库函数时，程序员首先必须了解被调用函数的参数传递方法，有关这方面的详细内容将在本书后面部分进行介绍。下面是求对数运算的 C 和汇编语言程序的两个例子。

【例 3.20】直接在 C 程序中调用库函数

```
#include   "math.h"
    main（）
    {
      float x,y;
    x=100.0;
    y=log（x）;
    }
```

【例 3.21】在汇编语言程序中调用库函数

```
.global   _log      ; 定义外部函数
LDF       1.0e2,R0  ; R0=100.0;
PUSHF     R0        ; 压入堆栈，传递参数
CALL      _log      ; 调用自然对数子程序
STF       R0,R1     ; 结果在 R0 中返回，并存入 R1 寄存器
```

用上述方法在 C 或汇编中调用库函数，对 C 或汇编程序进行编译或汇编形成目标模块，然后在链接时链入相应的函数库即可得到最后的可执行程序。

调用 C 编译器提供的库函数比较方便，且精度较高，缺点是运算量大，效率不高。这在运算量绰绰有余时当然不成问题，但是，当频繁调用库函数，特别是在一个循环次数较多的循环体内时，运算量就不容忽视了。因此，根据实际应用编写非线性函数的快速实现程序就显得极为必要。

与定点 DSP 芯片实现非线性函数类似，查表法也是浮点 DSP 芯片快速实现非线性函数的一种最基本的方法。但是，与定点数相比，浮点数的动态范围更大。如果不知道变量的动态范围，制表将非常困难。那么如何才能实现非线性运算的快速实现呢？

上面 3.5 节中介绍了定点芯片实现非线性运算的快速算法，实际上，这种思想更适合于浮点 DSP 芯片，因为利用浮点数据的格式以及浮点芯片提供的指令可以更为高效地实现。

3.6.2 节已经介绍，一个 TMS320C3x 的浮点数可以表示为尾数和指数的形式，而且在二进制表示时，尾数是一个归一化的值，其绝对值在 1~2 之间，而指数在－127~＋127 之间。因此，将一个浮点数首先转化为尾数和指数的形式可以便于用查表法实现非线性运算。一个浮点数 x 表示为 $x=m\times 2^e$，则 x 的常用对数为：

$$\lg(x)=\lg(m\times 2^e)=\lg(m)+\lg(2^e)=\lg(m)+e$$

因此，求 x 的对数实际上就变成了求尾数 m 的对数。由于求对数时 x 必须为正数，因此 m 的动态范围为 1~2，这样，对 m 作一个表格就比较方便，而且，表格不需要太大就可以获得较高的精度。

下面举一个浮点数开平方的例子来说明非线性函数的 DSP 快速实现方法。在下面的例子中，尾数直接采用查表法求得，表的大小为 10。直接采用查表法求尾数的非线性函数值时，为使总体的误差最小，在作表时应采用每一段的中点函数值作为表格中的函数值。对于大小为 10 的尾数开平方表，表中的值应是 1.05、1.15、1.25、1.35、1.45、1.55、1.65、1.75、1.85、1.95 这 10 个值的开平方值。

因为 $x=m\times 2^e$，因此 x 的开平方值应该是尾数 m 的开平方值乘以 2^e 的开平方值的乘积。如果 x 的指数 e 是一个偶数，且 $e1=e/2$，则 2^e 的开平方值为 2^{e1}；如果 e 是一个奇数，且 $e2=\lfloor e/2\rfloor$，式中符号⌊ ⌋是下取整，则 2^e 的开平方值为 $\sqrt{2}\cdot 2^{e2}$。例 3.22 是 DSP 快速开平方的 C 浮点模拟程序。

【例 3.22】DSP 快速实现浮点数开方的 C 模拟程序

```
#include <math.h>
#define     sqrt_2    1.414213562          /*2 的开平方值*/
 /*尾数的开平方表*/
float    sqrt_tab[10]={1.024695,1.072381,1.118034,1.161895,1.204159,
              1.244990,1.284523,1.322876,1.360147,1.396424};
float   sqrt_fast（float   xin）
{
```

```
    int    exponent, temp;
    float    mantissa, result1, result2;
    exponent = floor（log（xin）/log（2））; /*求 xin 的指数*/
    mantissa = xin/pow（2,exponent）          ; /求 xin 的尾数*/
    temp =  （int）（（mantissa-1）*10）    ; /*求查表的地址*/
    result1 = sqrt_tab[temp]                  ; /*查表得尾数的开方值*/
    temp = exponent/2;
    result1 = result1* pow（2,temp）          ; /*指数为偶数时的结果*/
    result2 = result1*sqrt_2                  ; /*指数为奇数时的结果*/
    if（（float）temp = = exponent/2.0 ）    return（result1）;
    else    return（result2）;
}
```

采用上面的开平方子程序实现浮点数 $x = 10000.0$ 的开平方的结果为 101.2。如果要提高运算的精度，一种方法是扩大尾数表的大小，另一种方法是采用上一章中介绍的混合法，在表格的每一段内采用折线近似法，适当增加运算量以换取精度的提高。如果两种方法并用，则完全可以满足大部分 DSP 应用对非线性运算的精度要求。

3.9　小　　结

本章讨论了 DSP 芯片进行定点运算所涉及的一些基本问题，包括数的定标、DSP 程序的定点模拟、DSP 芯片的定点运算以及定点实现非线性函数的快速实现方法等，充分理解这些问题对于用定点芯片实现 DSP 算法具有非常重要的作用。此外，本章还介绍了浮点 DSP 芯片中的数值表示格式，简要介绍了 DSP 浮点数的基本运算方法。

3.10　习题与思考题

（1）已知一十六进制数 3000H，若该数分别用 Q0、Q5、Q15 表示，计算该数的大小。

（2）若某一变量用 Q10 表示，计算该变量所能表示的数值范围和精度。

（3）已知 x=0.4567，试分别用 Q15、Q14、Q5 将该数转换为定点数（考虑舍入和不舍入两种情况）。

（4）函数 $f(x) = 2(1+x^2), -1 < x < +1$，为了保持最大精度，试确定定点运算时自变量 x 和函数 $f(x)$ 的 Q 值。

（5）两个数 x、y 分别为 0.45 和 1.97，试采用 16 位定点方法（保持最大精度），计算 x、y 之和及乘积，并比较定点和浮点之结果。

（6）采用定点方法，计算 ln（105.6）的值，并比较定点与浮点结果。

（7）一个浮点数在计算机中是如何表示的？

（8）IEEE 单精度浮点数格式是如何表示的？试用 IEEE 单精度浮点格式表示浮点数 12345.678。

（9）TMS320C3x 单精度浮点数格式是如何表示的？试用 TMS320C3x 单精度浮点格式表示浮点数 12345.678。

（10）如何可以实现非线性浮点运算的快速计算？

第 4 章　DSP 芯片的开发环境

4.1　引　　言

可编程 DSP 芯片的开发需要一整套完整的软硬件开发工具。通常，DSP 芯片的软件开发工具包括代码生成工具、代码调试工具、代码优化工具、代码分析工具四大类；硬件开发工具分为硬件设计工具和硬件调试工具。DSP 芯片开发的基本过程是利用 C/汇编语言在选定的芯片上实现所需的功能模块。根据功能需求分析，选择能满足需求的芯片，然后完成系统的功能模块划分，利用 C/汇编语言实现各自的模块，同时利用硬件开发工具调试好硬件平台，最后通过软件调试工具进行系统联调，完成系统设计。在系统设计时，可以利用 DSP 芯片软件开发工具进行软件仿真，以测试系统功能。图 4-1 给出了 DSP 芯片的开发流程。

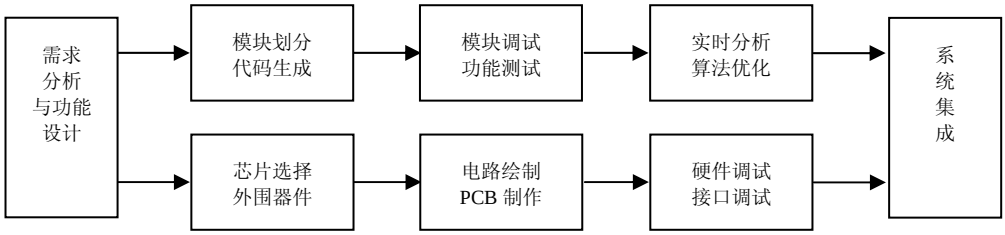


图 4-1　DSP 芯片的开发流程

本章将分别介绍软件开发环境和硬件开发环境，结合开发流程介绍各种开发工具。软件开发环境重点介绍 TI 公司的 Code Composer Studio（以下简称 CCS）集成开发环境。

4.2　软 件 开 发

系统需求分析完成后，要分别进行软件和硬件开发。软件开发首先要对功能模块进行进一步划分，便于代码实现；然后利用 C 语言、汇编语言实现各个功能模块；对模块分别进行语法级调试、算法级调试、系统级调试，通过性能分析工具和代码优化手段对调试结果进行进一步的分析和调整，使各功能模块整合成完整的系统，实现设计目标。

软件开发流程如图 4-2 所示。

源程序可以利用多种方式进行编辑。所形成软件的开发过程要使用代码生成工具、代码调试工具、代码优化工具、代码管理工具等软件开发工具。代码生成工具是指将用高级语言（如 C 语言）、汇编语言或用两种语言混合编写的 DSP 程序转换为可执行的 DSP 芯片目标代码的工具程序，主要包括 C 编译器、汇编器和链接器等。代码调试工具是对 DSP 程序及系统进行调试使之能够达到设计功能的工具程序，主要包括调试器。代码优化工具是对 C 语言程序进行优化使之能达到一定运算量要求的工具程序，主要包括优化器。代码管理工具是对整个系统所涉及的文件进行管理使之便于大规模系统开发的工具程序，主要包括工程管理器。

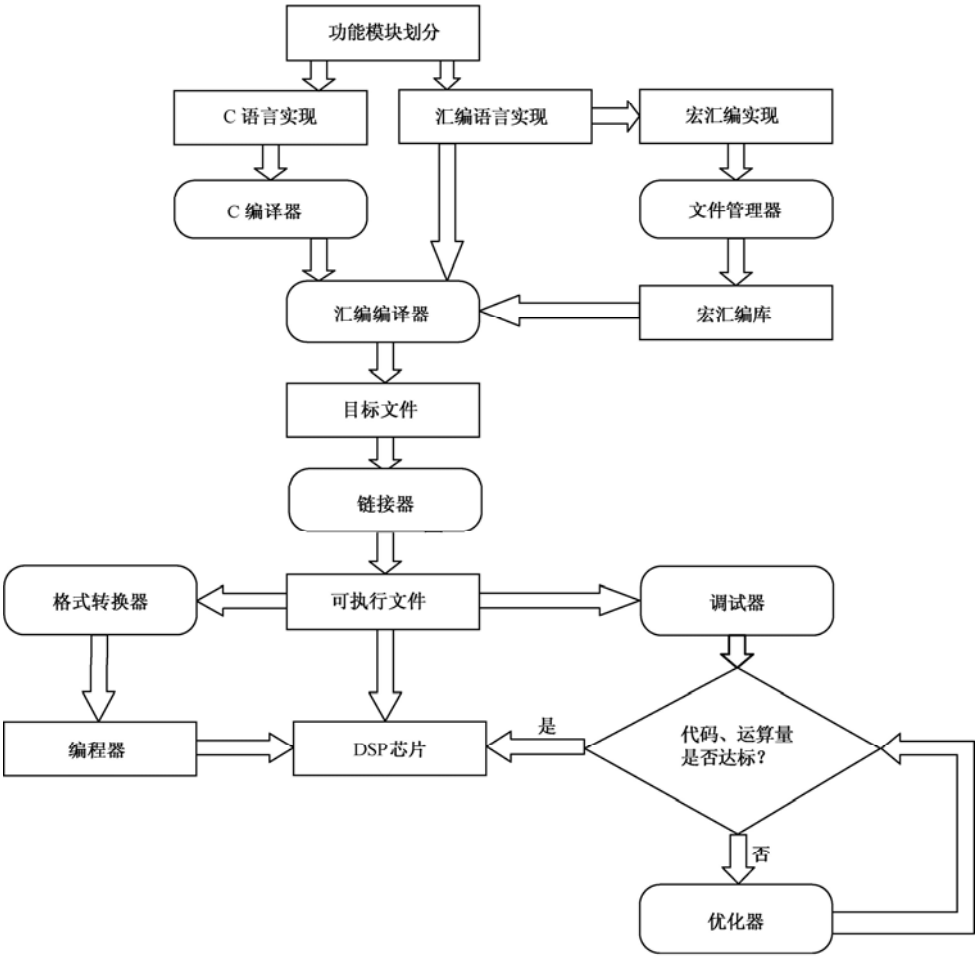


图 4-2　软件开发流程

早期的开发环境中，开发工具大多是以单个程序的形式出现，数目众多，不易操作，开发人员多采用批处理文件来处理开发过程中频繁使用的多个工具程序。表 4-1 列出了 TMS320C5000 V3.50 版工具程序。

表 4-1　TMS320C5000 V3.50 版工具程序

程序名	作　　用	程序名	作　　用
CL500.EXE	一步编译汇编链接程序 一步将.C 程序转换成.OUT 文件	AR500.EXE	文档管理程序 对目标文件库进行增加、删除、提取、替代等操作
AC500.EXE	C 文法分析程序 对.C 文件进行文法分析，生成.IF 中间文件	ASM500.EXE	COFF 汇编应用程序 将汇编语言程序转换为 COFF 目标文件.OBJ
OPT500.EXE	优化程序 对.IF 文件进行优化，生成.OPT 文件	HEX500.EXE	代码格式转换程序 将.OUT 文件转换为指定格式的文件
CG500.EXE	代码生成程序 将.IF 或.OPT 文件生成.ASM 文件	LNK500.EXE	链接程序 将目标文件链接成.OUT 文件
CLIST.EXE	交叉列表程序 对 CG500 生成的.ASM 文件进行交叉列表，生成.CL 文件	MK500.EXE	库生成应用程序

目前，各个 DSP 芯片生产商都推出了自己的集成开发环境，如 TI 公司的 CCS 集成开发环境等。开发人员可以借助集成开发环境方便、快速地完成开发流程。

4.3 软件开发环境

本节以 TI 公司推出的 DSP 芯片集成开发环境——CCS 为例，详细介绍 CCS 的安装、基本环境和使用方法等，结合软件开发流程，介绍各种开发工具。

1999 年，TI 推出 CCS 集成开发环境，这是 DSP 开发软件的一次革命性的变化。CCS 功能如图 4-3 所示，它采用 Windows 风格界面，集编辑、编译、链接、软件仿真、硬件调试及实时跟踪等功能于一体，集成的源代码编辑环境，使程序的修改更为方便；集成的代码生成工具，使开发设计人员不必在 DOS 窗口输入大量的命令及参数；集成的调试工具，使调试程序一目了然，大量的观察窗口使程序调试得心应手。这些特性极大地方便了 DSP 程序的设计和开发。

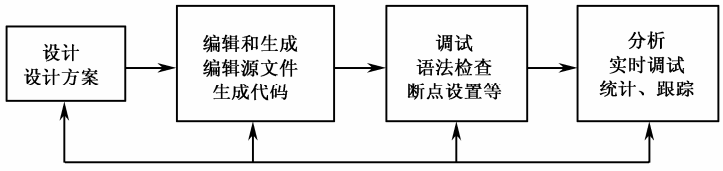


图 4-3　CCS 功能

更为重要的是，CCS 加速和增强了实时、嵌入信号处理的开发过程，它提供配置、构造、调试、跟踪和分析程序的工具，在基本代码产生工具的基础上增加了调试和实时分析的功能。开发设计人员可在不中断程序运行的情况下检查算法的正确性，实现对硬件的实时跟踪调试，从而大大缩短了程序的开发时间。

目前 CCS 软件已经历了 V1.0、V1.2、V2.0、V2.1、V2.2、V3.0、V3.1 和 V3.3 等版本，各个版本 CCS 软件的功能大体一致。V3.0 版本以前，CCS 只支持 TI 公司的一个 DSP 系列，每个系列都有对应的 CCS 版本，如 TMS320C5000 CCS V2.0、TMS320C6000 CCS V2.0 等，开发不同系列的 DSP 要安装对应的 CCS 软件。V3.0 及其后续版本 CCS 支持了所有 DSP 芯片系列，安装一个软件即可开发所有系列 DSP 芯片。目前使用较为广泛的是 CCS V2.2 版本，本节以 TMS320C5000 CCS V2.2 为例进行介绍。

4.3.1 CCS 安装与设置

CCS 对 PC 的最低要求为：操作系统 Microsoft Windows 98、Windows 2000 或 Windows XP，128M RAM，800M 空闲硬盘空间，Pentium 2 以上微处理器，SVGA 显示器（分辨率 800×600 以上）。建议使用 256M RAM 和 Pentium 3 以上的微处理器。

4.3.1.1 CCS 软件安装

将 CCS 安装光盘插入 CD-ROM 驱动器中，运行光盘根目录下的 setup.exe，按照安装向导的提示将 CCS 安装到硬盘中（系统默认的安装目录为 c:\ti）。安装完成后，安装程序将自动在计算机桌面上创建“CCS 2 （’C5000）”、“Setup CCS 2 （’C5000）”等几个快捷图标。

1．安装目录

CCS 安装完成后，将产生如图 4-4 所示的目录树结构，该目录树中包括以下目录：

- bin：应用程序目录；
- c5400：支持 54 系列芯片的工具目录；
- c5500：支持 55 系列芯片的工具目录；
- cc\bin：CCS 环境的程序文件目录；
- cc\gel：CCS 中使用的 GEL 文件目录；
- docs：当选择完全安装时将产生此目录，存放 CCS 的相关文档和手册；
- drivers：各种配置的驱动程序文件目录；
- examples：源代码例程目录；
- myprojects：CCS 默认存放用户设计文档的目录；
- plugins：各种工具程序的控件目录；
- specdig：第三方仿真器文件目录（需要安装第三方仿真器软件）；
- tutorial：CCS 使用教程目录；
- Uninstall：存放卸载 CCS 时需要的文件的目录。

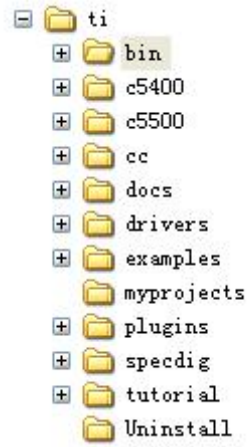


图 4-4　CCS 目录树结构

2．文件类型

CCS 使用以下几种类型的文件（以下文件名中的 Project、Program、Filename 等可由用户自行定义）：

- Project.pjt：CCS 使用的工程文件。CCS2.0 版本以前的工程文件名为 Project.mak；
- Program.c：C 程序源文件；
- Project.asm：汇编程序源文件；
- Filename.h：C 程序和 DSP/BIOS API 的头文件；
- Filename.lib：库文件；
- Project.cmd：链接命令文件；
- Program.obj：编译或汇编产生的目标文件；
- Program.out：完全编译、汇编和链接后产生的执行程序文件，在 CCS 中可以加载并运行这个执行程序；
- Program.wks：用于保存环境设置的工作文件；
- Program.cdb：配置数据库文件，此文件是调用 DSP/BIOS API 必需的文件，对其他应用程序而言同样也可使用。

当保存配置文件的同时还将产生以下三种文件：

（1）Programcfg.cmd：链接命令文件；

（2）Programcfg.h54：头文件；

（3）Programcfg.s54：汇编源文件。

3．环境变量

如果需要在命令行运行 CCS 的各种工具程序，请运行 c:\ti 目录下的 DosRun.bat 设置以下

环境变量：

- PATH：将 c:\ti\bin、c:\ti\cc\bin 等工具程序所在文件夹的绝对路径添加到系统路径中；
- C54x_A_DIR、C55x_A_DIR：汇编器查找 DSP/BIOS、RTDX 及代码产生工具所需库文件和头文件的搜索路径；
- C54x_C_DIR、C55x_C_DIR：编译器和链接器查找 DSP/BIOS、RTDX 及代码产生工具所需库文件和头文件的搜索路径；
- BSL5416_DIR：dsk5416 的库文件的绝对路径；
- TI_DIR：CCS 目录树中 ti 目录的绝对路径；
- C55X_CONFIG_FILE：配置文件 c55xx.cfg 的绝对路径。

在 Windows 98 或 Windows Me 系统中，运行 DosRun.bat 之前，需要确保系统的环境变量空间是足够的，否则会提示“Out of Environment Space”错误报警。如果出现报警，需运行下列命令：

COMMAND /E:2048

在目前计算机操作系统性能大大提高的情况下，一般不会出现上述问题。

4.3.1.2 CCS 软件设置

运行 CCS 软件之前，需首先运行 CCS 设置程序。CCS 设置程序用于建立 CCS 集成开发环境与目标板或 Simulator 之间的通信接口。CCS 是一个开放的环境，集成了 TI 公司的 Simulator 和 Emulator 的驱动程序，也可以通过设置不同的驱动完成对不同环境的支持，用户可直接使用 TI 的仿真器进行开发调试。如果使用的不是 TI 的仿真器，则要安装相应仿真器的驱动程序，下面以 SEED-XDS PP 型并口仿真器为例说明驱动程序的安装过程，同时以 CCS2.2 为例说明 CCS 下的驱动设置方法。

按照配套光盘的说明文件，运行安装文件（本例中为 setupcc54x.exe），按提示将驱动程序及相关设备的设置程序安装在计算机中，注意安装路径应与 CCS 的安装路径一致（默认路径为 c:\ti 目录）。

下面对 CCS 进行设置，假设安装路径为 c:\ti 目录。

（1）双击桌面上的“Setup CCS2（‘C5000）”图标，运行 CCS 设置程序；

（2）单击 Close 按钮关闭“Import Configuration”对话框（图 4-5）；

（3）执行菜单命令 Edit->Install Driver，选择图 4-6 中的 sdgo5xx.dvr 并打开；

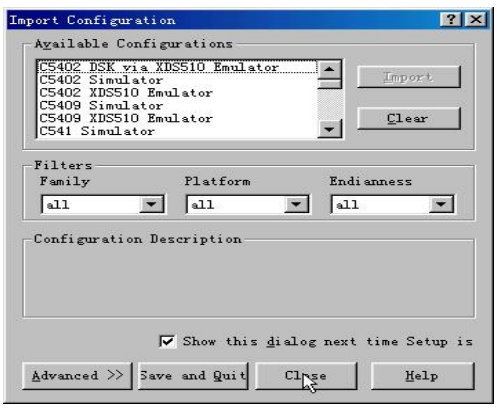


图 4-5　导入配置

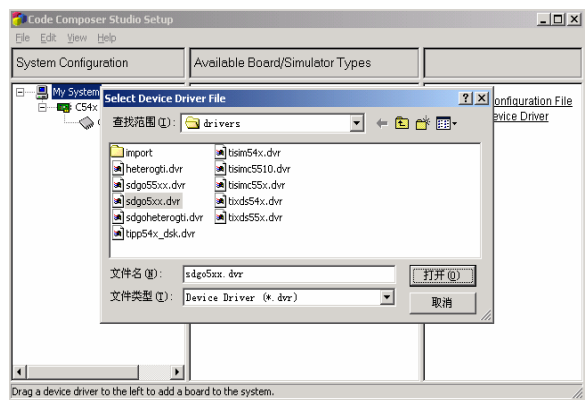


图 4-6　选择驱动程序文件

（4）在 Device Driver 栏为驱动程序命名，单击 OK 按钮关闭 Device Driver Properties 窗口；

（5）在 Available Board/Simulator Types 栏选中 sdgo5xx，执行菜单命令 Edit->Add to System（图 4-7）；

（6）如图 4-8 所示，单击 Board Properties 栏，将 I/O 端口值修改为 0x378（并行口 1）；

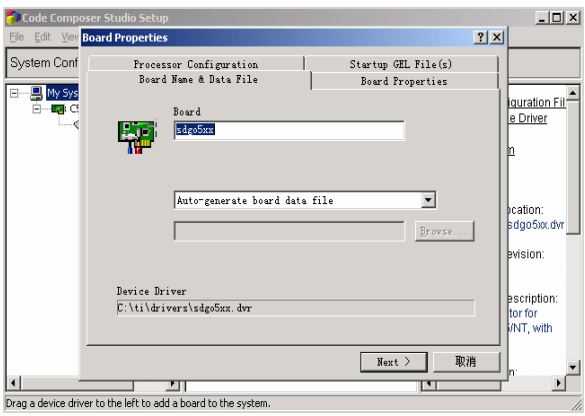


图 4-7　将驱动程序添加入系统

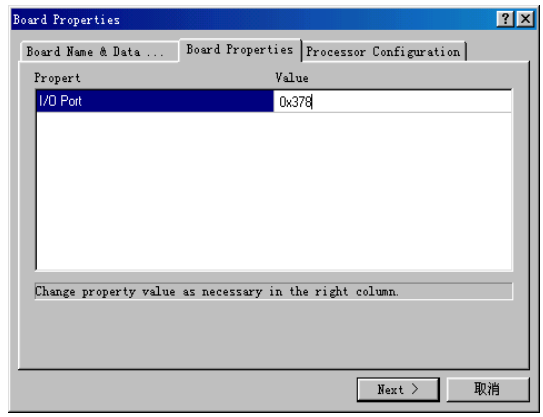


图 4-8　设置端口地址

（7）如图 4-9 所示，单击 Processor Configuration 栏，选中合适型号的处理器并单击 Add Single 按钮将 CPU_1 加入配置中，击 Next 按钮；

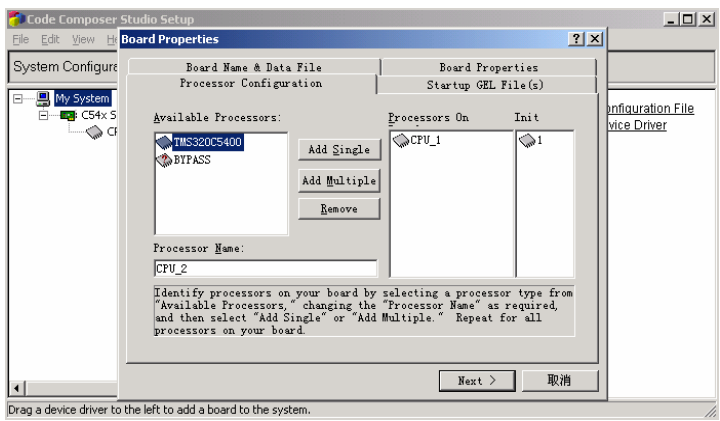


图 4-9　添加处理器

（8）单击 Startup GEL 栏的文件选择按钮，选择与开发板 DSP 芯片型号匹配的 GEL 文件，再单击 Finish 按钮结束；

（9）File->Save 保存设置，退出 Code Composer Setup 程序。

上述设置完成后，再双击桌面上的 SDConfig 图标，在左端树形目录下选择端口号 378，在双击 378 目录下的 Emu 目录，在右边的窗口设置硬件名和仿真并口类型。

以上的软件设置是针对 Emulator 而言的。在通常情况下，还应安装 Simulator 的驱动程序，此时只需执行 Edit->Add to System，将 C54x Simulator 添加到系统中即可使用 CCS 的 Simulator 功能。

4.3.2 CCS 基本环境

4.3.2.1 CCS 组件及其特征

如图 4-10 所示，CCS 由以下四部分组件构成：

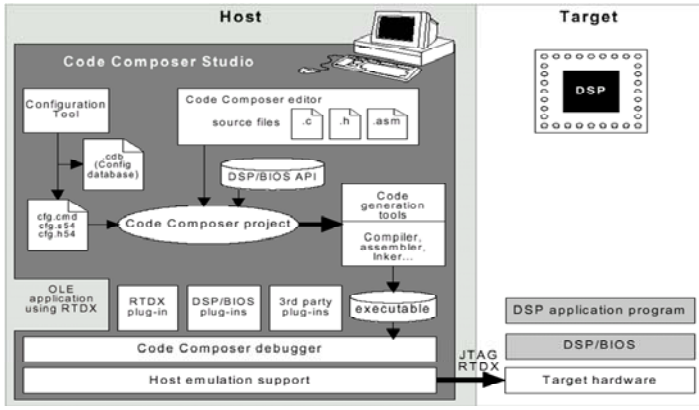


图 4-10　CCS 组件及工作机理

（1）TMS320C54x 代码产生工具；

（2）CCS 集成开发环境（IDE）；

（3）DSP/BIOS 插件及 API 函数；

（4）RTDX 插件，主机（host）接口及 API 函数。

1．代码产生工具

代码产生工具是 CCS 集成开发环境的基础。CCS 中集成了如下代码产生工具：

C 编译器（C Compiler）：对 C 语言源代码进行编译，产生汇编语言源文件。编译器还包括优化器（Optimizer），优化代码以提高程序性能。

汇编器（Assembler）：将汇编语言源文件转换为目标文件。TI 公司的汇编器创建的目标文件采用了一种称为 COFF 的目标文件格式（Common Object File Format，COFF）。采用这种目标文件格式更利于模块化编程，并且为管理代码段和目标系统存储器提供更强有力和更加灵活的方法。

链接器（Linker）：将目标文件链接成可执行文件。在基于 COFF 格式的目标文件支持下，开发人员编写汇编程序或高级语言程序时，不必为程序代码或变量指定目标地址，可由链接器对 COFF 目标文件和目标库进行重定位。这为程序编写和移植提供了极大的方便。

文档管理器（Archiver）：将一组文件集合到一个称为库文件的归档文件中，同时允许开发设计人员删除、替代、释放和添加库成员。

汇编翻译工具（Assembly Translator Utility）：将包含有助记符指令的汇编语言源文件转换为包含代数指令的汇编语言源文件。

库生成工具（Library-build Utility）：可以使用它构造定制的运行支持库。

运行支持库（Run-time-support Libraries）：包括 ANSI 标准运行支持函数、编译工具函数、浮点算术运算函数和 I/O 函数等 C 编译器支持的函数。

代码格式转换工具（Hex Conversion Utility）：将目标文件转换为 TI-Tagged、ASCII-HEX、

Intel、Motorola-s 或 Tektronix 目标格式，便于将转换后的文件编程到 EPROM 中。

交叉列表程序（Cross-reference Lister）：将链接的目标文件作为输入文件，产生交叉列表文件。列表文件中显示链接后源文件中的符号及其定义和引用情况。

绝对地址列表（Absolute Lister）：将链接后的目标文件作为输入文件，产生.abs 输出文件。对.abs 文件进行汇编就可产生一个列表文件，该文件中显示的是目标代码的绝对地址。

2．CCS 集成开发环境

CCS 集成开发环境可以编辑、编译、链接和调试 DSP 目标程序。

（1）程序编辑器（Editor）

CCS 允许编辑 C 和汇编源代码。如图 4-11 所示，程序编译连接后，还可以在 C 代码之后显示与之对应的汇编指令。集成编辑环境提供以下功能：

- 关键词、注释和字符串的高亮显示；
- 用圆括号和花括号对 C 代码块作标记，并可方便地查到与之配对的右括号；
- 自动缩进，自定义 Tab 键空格数；
- 书签，快速查找，在一个或多个文件中查找或替换；
- 多次 UNDO 和 REDO；
- 随时获取相关的在线帮助；
- 定制键盘命令。

（2）应用程序生成（Building）

与以往 DSP 开发软件不同的是，CCS 使用工程（Project）来管理应用程序设计文档，这与 Microsoft Visual Studio 有很大的相似之处。工程中可包括有 C 源代码、汇编源代码、目标文件、库文件、链接命令文件和头文件（图 4-12）。

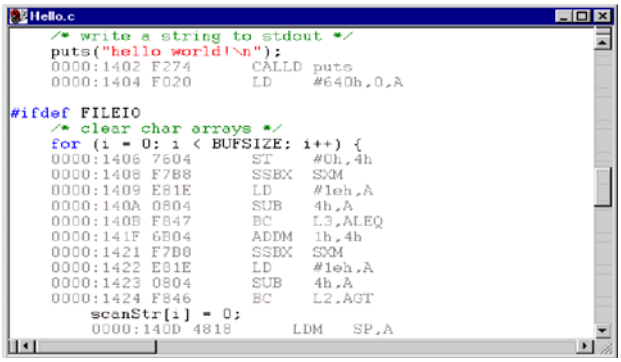


图 4-11　C 与汇编同时显示

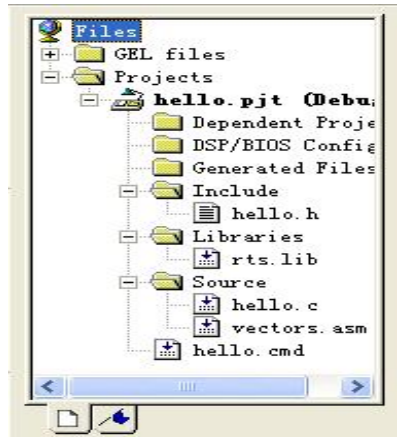


图 4-12　CCS 工程文件

在以往的开发工具中，编译、汇编和链接是三个独立的执行程序。为了完成程序的编译调试，开发设计人员需要熟悉每个程序的相关参数，且需在 DOS 窗口下输入这些繁琐的命令。在 CCS 集成开发环境下，只需在 Options 窗口中修改这些参数，编译连接可以一步完成。

使用工程文件，CCS 能自动查找工程所需的头文件并将之加入工程中。如果不想使用 CCS

的 Build 功能，还可使用传统的 makefile。

（3）程序调试（Debugging）

CCS 的调试工具具有以下功能：

- 设置一个或多个断点；
- 在断点处自动更新窗口；
- 使用 Watch 窗口察看变量；
- 查看和编辑存储器和寄存器；
- 使用 Probe Point 工具在主机与目标系统间传输数据流；
- 可对目标系统中的信号绘图显示；
- 使用 Profile Point 查看执行统计信息；
- 观察目标系统中执行的反汇编和 C 指令。

CCS 同时还提供 GEL 语言，允许开发者向 CCS 菜单中增加功能。

3．DSP/BIOS插件

在软件开发的分析阶段，传统的调试手段对诊断实时系统中的复杂问题无能为力。CCS 提供支持实时分析的 DSP/BIOS 插件，可以使用它来实时跟踪和监视 DSP 应用程序，同时对实时性能的影响达到最小。图 4-13 所示的执行图（Execution Graph）显示了不同线程的执行顺序。在 CCS 中，线程是一个通用概念，它可指任何执行的任务，如硬件中断服务（ISR）子程序、软件中断、任务及周期函数等。

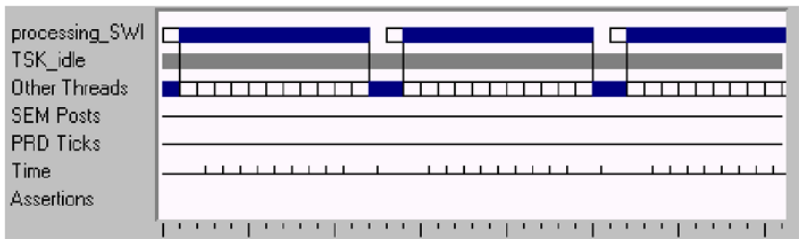


图 4-13　CCS 中的 Execute Graph

DSP/BIOS API 提供以下实时分析特性：

- 程序跟踪：显示写入目标日志的事件并在程序执行过程中反映动态控制流程；
- 性能监控：跟踪统计目标板资源的使用情况，如处理器负载和线程时序等；
- 文件流：将目标板上的 I/O 对象与主机上文件联系在一起。

DSP/BIOS 同时提供基于优先级的调度方案，支持函数的周期性执行和多优先级线程。

4．硬件仿真

TI DSP 芯片提供片上仿真支持，使 CCS 能控制程序运行并实时监视程序活动。计算机与 DSP 通信通过一个 JTAG 口完成，这种连接方式对 DSP 目标系统的实时性能没有太大的影响。硬件仿真器（如 TI XDS510）提供了与计算机通信的 JTAG 接口，DSP 板则提供板上的 JTAG 仿真接口。

片上仿真硬件提供以下功能：

- 开始、停止或复位 DSP；
- 将代码和数据加载到 DSP；

- 检查 DSP 中寄存器和存储器；
- 设置硬件指令或数据相关的断点；
- 各种计算功能，包括精确到指令周期的剖切（Profiling）功能；
- 提供主机和目标 DSP 间的实时数据交换。

5．实时数据交换（RTDX）

CCS 软件支持的一个重要仿真功能是 RTDX，RTDX 功能使计算机和 DSP 之间能双向实时通信，如图 4-14 所示。

RTDX 提供一个实时和连续的可视环境，使开发者能看到 DSP 应用程序工作的真实过程。RTDX 允许系统开发者在不停止运行目标应用程序的情况下在计算机和 DSP 芯片之间传输数据，同时还可在主机上利用对象链接嵌入（OLE）技术分析和观察数据。这样可以提供给开发者一个真实的系统工作过程，从而缩短开发时间。

RTDX 由目标板和计算机两部分构成。一个小的 RTDX 软件库运行在目标 DSP 上，开发者的 DSP 应用程序调用软件库的 API 函数，可从其接收数据或往其发送数据。软件库使用片内仿真硬件，通过增强型 JTAG 接口将数据发往计算机或从计算机接收数据。数据的传输是实时进行的，且不影响目标 DSP 的程序运行。

计算机平台提供一个 RTDX 库配合 CCS 工作。显示和分析工具可以通过一个简单易用的 COM API 与 RTDX 通信，从目标板获得数据或将数据发往 DSP。设计者可使用标准软件显示包，如 National Instruments 的 LabVIEW，Quinn-Curtis 的实时图形工具或 Microsoft Excel，也可开发自己的 VB 或 VC 应用程序。

RTDX 还可录制实时数据并回放做非实时的分析。图 4-15 为采用 LabVIEW 做非实时分析的实例：假设目标板存在一个原始信号和一个经 FIR 滤波器滤波后的信号，这两个信号均通过 RTDX 送往计算机。在主机方，LabVIEW 通过 RTDX COM API 获得信号数据，这两个信号显示在图中左半部分。为验证滤波器功能，使用 LabVIEW 软件产生一个功率谱波形显示在图中右半部分。滤波信号功率谱显示在右上角，目标板上的原始信号通过一个 LabVIEW FIR 滤波器进行软件仿真，其功率谱显示在右下角。比较这两个功率谱即可判别 FIR 滤波器算法正确。

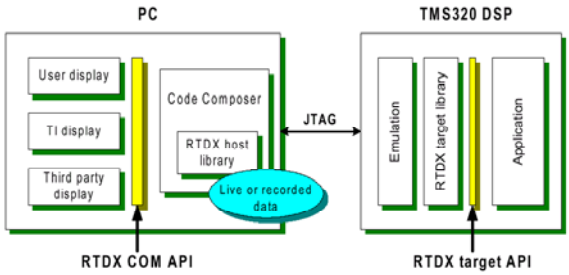


图 4-14　RTDX 工作机理

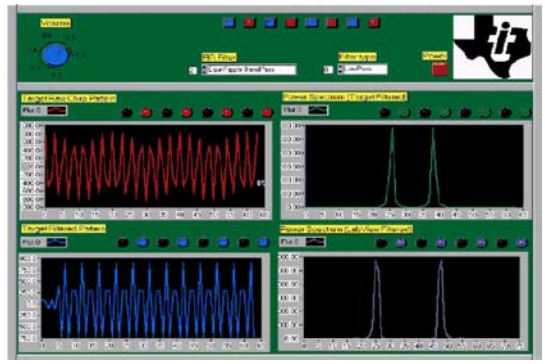


图 4-15　LabView 显示与分析

RTDX 非常适合应用在控制系统、伺服系统和语音处理等场合。如无线应用厂商可以捕获声码器的输出以检查语音处理算法是否正确；硬盘驱动器设计人员无需往伺服电机上加不正确的信号（如此可能损坏硬盘）就可以测试其产品；引擎控制设计人员可以在控制器正常运行时分

析温度及其他环境变化带来的影响。在这些应用中，设计者可选择可视工具显示对调试和分析有用的信息。未来的 TI DSP 芯片将使 RTDX 带宽增加，提供大型程序的可视控制和管理。

此外，第三方软件开发者可以提供 ActiveX 插件以扩充 CCS 的功能。

4.3.2.2 CCS 集成环境的使用

使用 CCS 开发 DSP 应用程序首先应熟悉 CCS 集成开发环境。在对应用系统进行硬件调试之前，设计者可使用 Simulator 在没有目标板的情况下模拟 DSP 程序的运行。在 CCS 软件中，Simulator（软件模拟器）与 Emulator（硬件仿真器）使用的是相同的集成开发环境，因此熟悉 CCS 集成开发环境可从 Simulator 开始。

如果系统中只安装了 Simulator 的驱动程序，则运行 CCS 即可启动 Simulator 的运行；如果系统中同时安装了 Simulator 和 Emulator 的驱动程序，则运行 CCS 时将启动并行调试管理器（Parallel Debug Manager）（如图 4-16 所示），此时需从菜单中选择 Open->C54xx Simulator 以启动 Simulator 的运行，出现与图 4-17 类似的窗口。

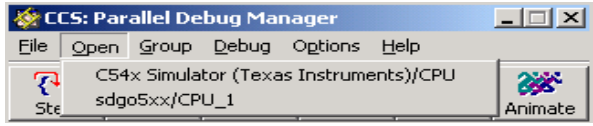


图 4-16　并行调试管理器

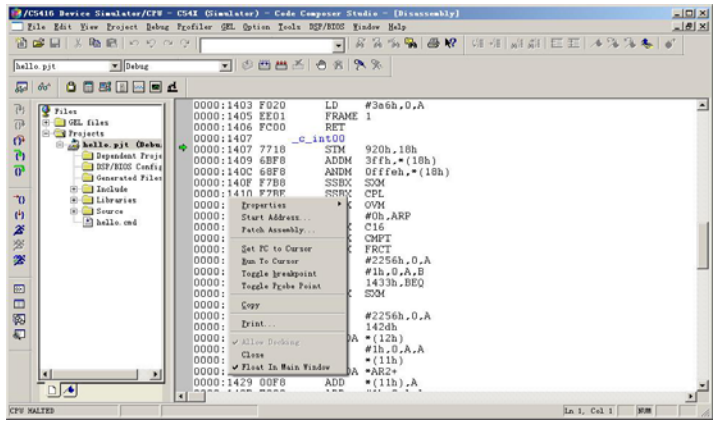


图 4-17　CCS 运行主窗口

在 CCS 集成开发环境中，除 Edit 窗口外其余所有窗口和所有的工具栏都是可定位（Allow Docking）的，也就是说可将这些窗口和工具栏拖至屏幕的任何位置（包括移至主窗体之外）。

在 CCS 中，所有的窗口都支持内容相关菜单（Context Menu）。在窗口内单击鼠标右键即可弹出内容相关菜单，菜单中包含有与该窗口相关的选项和命令。

1．菜单

在 CCS 集成开发环境中共有 12 项菜单，下面就其中较为重要的菜单功能加以介绍。

（1）File 菜单

File 菜单提供了与文件操作有关的命令。除 Open/Save/Print/Exit 等常见的命令外，File 菜

单还列出了以下几种文件操作命令，如表 4-2 所示。

表 4-2　File 菜单

菜单命令		功　　能
New	Source File	新建一个源文件（.c, .asm, .h, .cmd, .gel, .map, .inc 等）
	DSP/BIOS Config	新建一个 DSP/BIOS 配置文件
	Visual Linker Recipe	打开一个 Visual Linker Recipe 向导
	ActiveX Document	在 CCS 中打开一个 ActiveX 文档（如 Microsoft Word 或 Microsoft Excel 等）
Load Program		将 COFF（.out）文件中数据和符号加载入实际或仿真目标板（Simulator）中
Load Symbol		当调试器不能或无需加载目标代码（如目标代码存放于 ROM 中）时，仅将符号信息加载入目标板；此命令只清除符号表，不更改存储器内容和设置程序入口
Reload Program		重新加载 COFF 文件，如果程序未作更改则只加载程序代码而不加载符号表
Load GEL		在调用 GEL 函数之前，应将包含该函数的文件加载入 CCS 中，以将该 GEL 函数调入内存；当加载的文件修改后，应先卸掉该文件，再重新加载该文件，从而使修改生效 加载 GEL 函数时将检查文件的语法错误，但不检查变量是否已定义
Data	Load	将 PC 文件中的数据加载入目标板，可以指定存放的地址和数据的长度；数据文件可以是 COFF 文件格式，也可以是 CCS 支持的数据格式
	Save	将目标板存储器数据存储到一个 PC 数据文件中
Work space	Load Workspace	将工作空间文件加载到当前工作界面
	Save Workspace	将当前工作界面的状态，包括窗口的大小位置等信息保存到工作空间文件
	Save Workspace As	将当前工作界面的状态，另存为一个工作空间文件
File I/O		CCS 允许在 PC 文件和目标 DSP 之间传送数据；一方面可从 PC 文件中取出样值用于模拟，另一方面也可将目标 DSP 处理的数据保存到计算机文件中 File I/O 功能应与 Probe Point 配合使用；Probe Point 将告诉调试器在何时从 PC 文件中输入或输出数据 File I/O 功能并不支持实时数据交换，实时数据交换应使用 RTDX
Launch Setup		运行 CCS Setup 程序
Close Session		退出当前工作界面

（2）Edit 菜单

Edit 菜单提供的是与编辑有关的命令。除了 Undo/Redo/Delete/Select All/Find/Replace 等常用的文件编辑命令外，CCS 还支持以下几种编辑命令，如表 4-3 所示。

表 4-3　Edit 菜单

菜单命令		功　　能
Find in Files…		在多个文本文件中查找特定的字符串或表达式
Go To…		快速跳转到源文件中某一指定行或书签处
Memory	Edit	编辑某一存储单元
	Copy	将某一存储块（标明起始地址和长度）数据拷贝入另一存储块
	Fill	将某一存储块填入某一固定值
	Patch Asm	在不修改源文件的情况下修改目标 DSP 特定地址的执行代码
Register…		编辑指定的寄存器值，包括 CPU 寄存器和外设寄存器；由于 Simulator 不支持外设寄存器，因此不能在 Simulator 下监视和管理外设寄存器内容
Variable…		修改某一变量值；对 TI 定点 DSP 芯片而言，如果目标 DSP 由多个页面构成，则可使用@prog, @data 和@io 来分别指定程序区、数据区和 I/O
Command Line…		提供输入表达式或执行 GEL 函数的快捷方法
Column Editing		选择某一矩形区域内的文本进行列编辑（剪切、复制及粘贴等）
Enable Extenal Editor		使能外部文本编辑器
Bookmarks…		在源文件中定义一个或多个书签便于快速定位；书签被保存在 CCS 的工作区（Workspace）内以便随时被查找到

（3）View 菜单

在 View 菜单中，可以选择是否显示 Standard 工具栏、GEL 工具栏、Project 工具栏、Debug 工具栏、Editor 工具栏和状态栏（Status bar）。此外 View 菜单中还包括如表 4-4 所示的显示命令。

表 4-4　View 菜单

菜单命令		功　　能
Disassembly		当将程序加载入目标板后，CCS 将自动打开一个反汇编窗口。反汇编窗口根据存储器的内容显示反汇编指令和符号信息
Memory…		显示程序空间、数据空间或 IO 空间指定地址的内容
CPU Registers	CPU Register	显示 DSP 的寄存器内容
	Peripheral Regs	显示外设寄存器内容；Simulator 不支持此功能
Graph	Time/Frequency	在时域或频域显示信号波形；频域分析时将对数据进行 FFT 变换，时域分析时数据无需进行预处理；显示缓冲的大小由 Display Data Size 定义
	Constellation	使用星座图显示信号波形；输入信号被分解为 X、Y 两个分量，采用笛卡儿坐标显示波形；显示缓冲的大小由 Constellation Points 定义
	Eye Diagram	使用眼图来量化信号失真度；在指定的显示范围内，输入信号被连续叠加并显示为眼睛的形状
	Image	使用 Image 图来测试图像处理算法；图像数据基于 RGB 和 YUV 数据流显示
Tasks	Refresh Tasks	CCS 1.2 版、2.0 版和 2.2 版均不支持
	Not Available	CCS 1.2 版、2.0 版和 2.2 版均不支持
Watch Window		用来检查和编辑变量或 C 表达式，可以以不同格式显示变量值，还可显示数组、结构或指针等包含多个元素的变量
Quick Watch		快速查看和修改变量，添加观察变量到 Watch 窗口
Call Stack		检查所调试程序的函数调用情况；此功能仅在调试 C 程序时有效，且程序中必须有一个堆栈段和一个主函数，否则将显示“C source is not available.”
Expression List		所有的 GEL 函数和表达式都采用表达式求值程序来估值；求值程序可对多个表达式求值，在求值过程中可选择表达式并按 Abort 按钮取消求值；这项功能在 GEL 函数执行到死循环或执行时间太长时有用
Output Window		标准输出窗口
Project		CCS 启动后将自动打开工程视图；在工程视图中，文件按其性质分为源文件、头文件、库文件及命令文件
Mixed Source/Asm		同时显示 C 代码及相关的反汇编代码（位于 C 代码下方）
Realtime Refresh Options		CCS 1.2 版、2.0 版和 2.2 版均不支持

在 View 菜单中，Graph 是一个很有用的功能，它可以逼真地显示信号波形。在 Graph 窗口中使用了两个缓冲器：获取缓冲和显示缓冲。获取缓冲驻留在实际或仿真的目标板上，它保存有你感兴趣的数据。当图形更新时，获取缓冲从实际或仿真的目标板读取数据并更新显示缓冲。显示缓冲则驻留在主机存储器中，它记录了历史数据。波形图则是根据显示缓冲的数据绘制的。

当输入所需的参数并确认后，Graph 窗口从获取缓冲中接收指定长度（由 Acquisition Buffer Size 定义）和指定起始地址（由 Start Address 定义）的 DSP 数据。

（4）Project 菜单

CCS 使用工程来管理设计文档。CCS 不允许直接对汇编或 C 源文件 Build 生成 DSP 应用程序，只有在建立工程文件的情况下 Project 工具栏上的 Build 按钮才会有效。工程文件被存盘为.pjt 文件。在 Project 菜单中包括一些常见的命令如 New/Open/Close 等，此外还包括如表 4-5 所示的菜单命令。

（5）Debug 菜单

Debug 菜单显示常用的调试命令，如表 4-6 所示。

表 4-5　Project 菜单

菜单命令	功　　能
Add Files to Project…	CCS 根据文件的扩展名将文件添加到工程的相应子目录中；工程中支持 C 源文件（*.c*）、汇编源文件（*.a*, *.s*）、库文件（*.o*, *.lib）、头文件（*.h）和链接命令文件（*.cmd）；其中 C 和汇编源文件可被编译和链接；库文件和链接命令文件只能被链接；CCS 会自动将头文件添加到工程中
Use External Makefile...	使用外部的 Makefile 文件对工程进行编译、链接设置
Export to Makefile…	将当前工程的编译、连接设置输出到 Makefile 文件
Compile File	对 C 或汇编源文件进行编译
Build	重新编译和链接；对于那些没有修改的源文件，CCS 将不重新编译
Rebuild All	对工程中所有文件重新编译并链接生成输出文件
Stop Build	停止正在 build 的进程
Build Clean	清除编译链接时产生的中间文件，主要是目标文件
Configurations…	选择当前工程的编译、链接的配置文件
Build Options…	用来设定编译器、汇编器和链接器的参数
Project Dependencies…	设置工程之间的依赖关系，针对大型系统中的多个子系统
ShowProject Dependencise	显示工程之间的依赖关系
Show File Dependencies Scan All File Dependencies	为了判别哪些文件应重新编译，CCS 在 build 一个程序时会生成一个关系树（Dependency Tree）以判别工程中各文件的依赖关系；使用此二菜单命令则可以观察工程的关系树
Recent Project Files	加载最近打开的工程文件

表 4-6　Debug 菜单

菜单命令	功　　能
Breakpoints…	设置断点，程序在执行到断点时将停止运行；当程序停止运行时，可以检查程序的状态，查看和更改变量值，查看堆栈等；在设置断点时应注意以下两点： 1．不要将断点设置在任何延时分支或调用的指令处； 2．不要将断点设置在 repeat 块指令的倒数一二行指令处
Probe Points…	允许更新观察窗口并在算法的指定处（设置 Probe Point 处）将 PC 文件数据读至存储器或将存储器数据写入 PC 文件中，此时应设置 File I/O 属性 对每一个建立的窗口，默认情况是在每个断点（Breakpoint）处更新窗口显示，然而也可以将其设置为到达 Probe Point 处时更新窗口 使用 Probe Point 更新窗口时，目标 DSP 将临时中止运行；当窗口更新后，程序继续执行；因此 Probe Point 不能满足实时数据交换（RTDX）的需要
StepInto	单步运行；如果运行到调用函数处将跳入函数单步执行
StepOver	执行一条 C 或汇编指令；与 StepInto 不同的是，为保护处理器流水线，该指令后的若干条延时分支或调用将同时被执行；如果运行到函数调用处将执行完该函数而不跳入函数执行，除非在函数内部设置了断点
StepOut	如果程序运行在一个子程序中，执行 StepOut 将使程序执行完该子程序回到调用该函数的地方；在 C 源程序模式下，根据标准运行 C 堆栈来推断返回地址，否则根据堆栈项的值来求得调用函数的返回地址；因此，如果汇编程序使用堆栈来存储其他信息，则 StepOut 命令可能工作不正常
Run	从当前程序计数器（PC）执行程序，碰到断点时程序暂停执行
Halt	中止程序运行
Animate	运行程序；碰到断点时程序暂停运行，在更新未与任何 Probe Point 相关联的窗口后程序继续运行；动画（Animate）命令的作用是在每个断点处显示处理器的状态，可以在 Option 菜单下选择 Animate Speed 来控制动画的速度
Run Free	忽略所有断点（包括 Probe Point 和 Profile Point），从当前 PC 处开始执行程序。此命令在 Simulator 下无效；使用 Emulator 进行仿真时，此命令将断开与目标 DSP 的连接，因此可移走 JTAG 或 MPSD 电缆；在 Run Free 时还可对目标 DSP 硬复位
Run to Cursor	执行到光标处，光标所在行必须为有效代码行
Set PC to Cursor	将鼠标所在处的程序地址设置到 PC 寄存器中
Multiple Operation…	设置单步执行的次数
Assembly/Source Stepping	针对 C 语言或汇编语言进行 step Into、step Over 操作

（续表）

菜单命令	功　　能
Reset DSP	重新初始化所有寄存器到其上电状态并中止程序运行
Restart	将 PC 值恢复到程序的入口；此命令并不开始程序的执行
Go Main	在程序的 main 符号处设置一个临时断点；此命令在调试 C 程序时起作用
Reset Emulator	重新初始化硬件仿真器
Always Connect at Startup	选择调试器是否在程序启动时运行
Enable Task Level Debugging	CCS 1.2 版、2.0 版和 2.2 版均不支持
Real Time Mode	CCS 1.2 版、2.0 版和 2.2 版均不支持
Enable Rude Real Time	CCS 1.2 版、2.0 版和 2.2 版均不支持

（6）Profiler 菜单

剖切（Profiling）是 CCS 的一个重要功能。它可提供程序代码特定区域的执行统计，使开发设计人员能检查程序的性能，从而可对源程序进行优化设置。使用剖切功能可以观察 DSP 算法占用了多少 CPU 时间，还可以用它来剖切处理器的其他事件，如分支数、子程序调用次数及中断发生次数等。菜单命令如表 4-7 所示，相关的剖切时钟设置和剖切统计数据分别见图 4-18 和图 4-19。

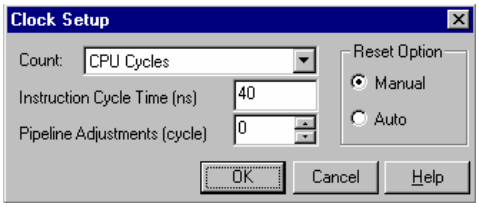


图 4-18　时钟设置

表 4-7　Profiler 菜单

菜单命令	功　　能
Enable Clock	为了获得指令周期及其他事件的统计数据，必须使能剖切时钟（Profile Clock）。当剖切时钟被禁用时，将只能计算到达每个剖切点的次数，而不能计算统计数据 剖切时钟被作为一个变量（CLK）通过 Clock 窗口被访问；CLK 变量可在 Watch 窗口观察，并可在 Edit Variable 对话框内修改其值；CLK 还可在用户定义的 GEL 函数中使用 指令周期的计算方式与使用的 DSP 驱动程序有关；对使用 JTAG 扫描路径进行通信的驱动程序，指令周期通过处理器的片内分析功能进行计算，其他的驱动程序则可能使用其他类型的定时器；Simulator 使用仿真的 DSP 片内分析接口来统计剖切数据；当时钟使能时，CCS 调试器将占用必要的资源实现指令周期的计数
Clock Setup…	设置时钟；在 Clock Setup 对话框中（图 4-18 所示），Instruction Cycle Time 域用于输入执行一条指令的时间，其作用是在显示统计数据时将指令周期数转换为时间或频率用于显示 在 Count 域选择剖切的事件；对某些驱动程序而言，CPU Cycles 可能是唯一的选项；对于使用片内分析功能的驱动程序而言，可以剖切其他事件，如中断次数、子程序或中断返回次数、分支数及子程序调用次数等 可使用 Reset Option 参数来决定如何计数；如选择 Manual 选项，则 CLK 变量将不断累加指令周期数，这与 TI Simulator 类似；如选择 Auto 选项，则在每次 DSP 运行前将自动将 CLK 置为 0，因此 CLK 变量显示的是上一次运行以来的指令周期数，这与 TI Emulator 类似
View Clock	打开 Clock 窗口，显示 CLK 变量的值；双击 Clock 窗口的内容可直接将 CLK 变量复位
Start New Session..	开始一个新的剖切会话，如图 4-19

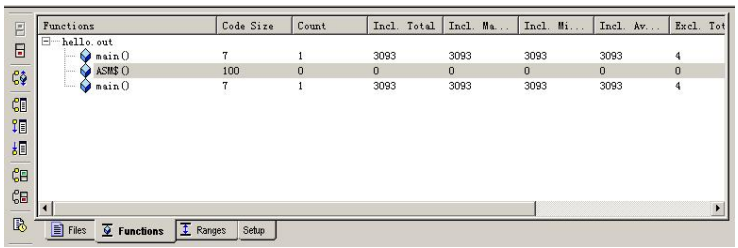


图 4-19　剖切统计数据

（7）GEL 菜单

C5000 CCS 软件本身提供 C54x 和 C55x 的 GEL 函数，它们在 c5000.gel 文件中定义。表 4-8 列出了 c5000.gel 文件中定义的 GEL 函数。用户还可将常用的 GEL 函数添加到 GEL 菜单中，此时需使用 menuitem 关键词在 GEL 菜单下建立一个新的下拉菜单项。此外还可使用 dialog 和 slider 关键词建立对话框和滑动条对象。

表 4-8　GEL 菜单

菜单命令		功　　能
C55x		提供 C55xReset 函数用于复位 C55x DSP 芯片
C54x	C54x_CPU_Reset	复位目标 DSP、复位存储器映射、禁止存储器映射及初始化寄存器
	C541_Init C542_Init C543_Init C545_Init C546_Init C548_Init C549_Init	对 C54x DSP 复位 使能存储器映射 设置指定 DSP 芯片的存储器映射
	C5402_Init C5409_Init C5410_Init C5416_Init C5420_Init C5421_Init C5402_DSK_Init	对 C54x DSP 复位 复位外设 使能存储器映射 设置指定 DSP 芯片的存储器映射

（8）Option 菜单

Option 菜单提供 CCS 的一些设置选项，如颜色、字体和键盘等。表 4-9 列出了 Option 菜单命令。

表 4-9　Option 菜单

菜单命令	功　　能
Font…	设置集成环境的字体
Disassembly Style…	设置反汇编窗口显示模式，包括反汇编成助记符或是代数符号、直接寻址与间接寻址用十进制、二进制或是十六进制显示
Customizer	设置集成开发环境；包括调试属性设置、搜索路径设置、视图颜色设置、编辑器设置、键盘命令设置、程序加载设置、控制窗口显示设置、最近访问文件设置等
Memory Map	用来定义存储器映射；存储器映射指明了 CCS 调试器能访问哪段存储器，不能访问哪段存储器；典型情况下，存储器映射与命令文件的存储器定义相一致

在 Option 菜单中，存储器映射（Memory Map）是一个重要的概念，有必要对其进行详细说明。

- 添加一个新的存储器映射范围

（a）选择 Option->Memory Map，将弹出 Memory Map 对话框。

（b）在对话框中选中 Enable Memory Mapping 以使能存储器映射。第一次运行 CCS 时，存储器映射即呈禁用状态（未选中 Enable Memory Mapping），也就是说 CCS 调试器可存取目标板上所有可寻址的存储器（RAM）。当使能存储器映射后，CCS 调试器将根据存储器映射设置检查其可以访问的存储器。如果要存取的是未定义或保护区数据，则调试器将显示默认值而不是存取目标板上数据。

（c）选择需修改的页面（Program、Data 或 IO）。如果程序只使用一个存储器页面，则可

以跳过这一步。

（d）按照命令文件的存储器定义，在 Starting address 域输入起始地址，在 Length 域输入存储器长度，在 Attributes 域选择存储器的读/写属性，再单击 Add 按钮即添加一个新的存储器映射范围。

CCS 调试器允许添加一个与已有的存储器范围有所重叠的新的存储器范围，此时重叠区域的读/写属性会作相应修改。

当定义好一个新的存储器范围后，如果想更改其读/写属性，则可以定义一个新的存储器范围（与该存储器具有相同起始地址和相同范围）并击 Add 按钮加入，则可将原存储器属性更改为新定义存储器属性。

- 添加一个新的存储器映射范围

将一个已有存储器映射属性设为 None-No Memory/Protected，可将该存储器范围删除。另一个删除存储器范围的方法是在 Memory Map 列表框内选中需删除的存储器范围，单击 Delete 按钮将其删除。

- 存取一个不合法的存储地址

当设计人员想读取一个被存储器映射保护的存储空间时，调试器将不从目标板读取数据，而是读取一个保护数据，通常为 0。因此一个非法的存储地址值通常显示为 0。也可在 Protected Value 域输入另外一个值，如 0XDEAD，这样当试图读取一个非法存储地址时将清楚地给予提示。

在判断一个存储地址是否合法时，CCS 调试器并不根据硬件结构作出比较结果。因此，调试器不能防止程序存取一个不存在的存储地址。

定义一个非法的存储器映射范围最好的方法是使用 GEL 嵌入函数，在运行 CCS 时自动执行。

（9）Tools 菜单

提供常用的工具集，如表 4-10 所示。

表 4-10　Tools 菜单

菜单命令	功　　能
Data Converter Support	使开发者能快速配置与 DSP 芯片相连的数据转换器
C54xx McBSP	使开发者能观察和编辑 McBSP 的内容；C54x DSP 芯片有多个高速、全双工的多信道缓冲串行口（McBSP），使该 DSP 芯片能直接与系统中的其他器件接口；McBSP 建立在标准串行接口的基础之上
C54xx Emulator Analysis	使开发者能设置和监视事件和硬件断点的发生；C54x 器件有一个片内分析模块，使用这些模块，可以计算特定硬件功能发生的次数或设置相应的硬件断点
C54xx DMA	使开发者能观察和编辑 DMA 寄存器的内容
C54xx Simulator Analysis	使开发者能设置和监视事件的发生；此工具为加载调试器使用的特定伪寄存器集提供了一个透明的手段；调试器使用这些伪寄存器存取片内分析模块
Command Window	在 CCS 调试器中输入所需的命令，输入的命令遵循 TI 调试器命令语法格式；由于许多命令都接受 C 表达式作为命令参数，从而使得指令集相对较小且功能较强；在命令窗口中输入 HELP 并回车可得到命令窗口支持的调试命令列表
Symbol Browser	显示加载的.out 文件中所有相关的文件、函数、全局变量、数据类型、标号等
Port Connect	将 PC 文件与存储器（端口）地址相连接，从而可从文件中读取数据，或将存储器（端口）数据写入文件中
Pin Connect	用于指定外部中断发生的间隔时间，从而使用 Simulator 来仿真和模拟外部中断信号 1．创建一个数据文件以指定中断间隔时间（用 CPU 时钟周期的函数来表示） 2．从 Tools 菜单下选择 Pin Connect 命令 3．单点 Connect 按钮，选择创建好的数据文件，将其连接到所需的外部中断引脚 4．加载并运行程序
Linker Configuration	使用 Visual Linker 链接程序
Algorithm Standard	用于产生用于标准算法的文档模板
RTDX	实时数据交换功能，使开发者能在不影响程序执行的情况下分析 DSP 程序的执行情况

（10）DSP/BIOS 菜单

提供常用的 BIOS 实时分析工具集，使开发者能对程序进行实时跟踪和分析。菜单命令如表 4-11 所示。

表 4-11　DSP/BIOS 菜单

菜单命令	功　　能
RTA Control Panel	打开实时分析工具控制面板，可以设置实时分析的相关参数，使能各种跟踪器
Execution Graph	调用执行图分析工具，打开执行图窗口，该窗口显示程序中各线程的运行情况
Statistics View	打开统计视图窗口，该窗口显示统计模块的实时数据
Message Log	打开信息日志窗口，该窗口显示日志模块传送的信息
Kernel/Object View	打开内核/模块窗口，该窗口显示当前程序中各种 BIOS 模块的实时配置、状态等信息；显示的模块包括 KNL 模块、TSK 模块、MBX 模块、SEM 模块、MEM 模块、SWI 模块
Host Channel Control	打开主机信道控制窗口，该窗口显示当前程序中定义的主机信道模块的相关信息
CPU Load Graph	打开 CPU 负载图窗口，该窗口显示目标板 CPU 的正在处理的负载信息

2．工具栏

CCS 集成开发环境提供六种工具栏，分别为 Standard Toolbar、GEL Toolbar、Project Toolbar、Edit Toolbar、Debug Toolbar 和 Plug-in Toolbar。这六种工具栏可在 View 菜单下选择是否显示。

（1）Standard Toolbar

如图 4-20 所示，标准工具栏包括以下常用工具：



图 4-20　Standard 工具栏

- New：新建一个文档；
- Open：打开一个已存在的文档；
- Save：保存一个文档，如尚未命名，则打开 Save As 对话框；
- Cut：剪切；
- Copy：复制；
- Paste：粘贴；
- Undo：取消上一次编辑操作；
- Undo History：显示取消操作的历史记录；
- Redo：恢复上一次编辑操作；
- Redo History：显示重做操作的历史记录；
- Find Next：查找下一个；
- Find Previous：查找上一个；
- Search Word：查找指定的文本；
- Find in Files：在多个文件中查找；
- Print：打印；
- Help：获取特定对象的帮助。

（2）GEL Toolbar

GEL 工具栏提供了执行 GEL 函数的一种快捷方法。如图 4-21 所示，在工具栏的左侧文本

输入框中输入 GEL 函数名，再单击右侧的执行按钮即可执行相应的函数。如果不使用 GEL 工具栏，也可以使用 Edit 菜单下的 Edit Command Line 命令执行 GEL 函数。



图 4-21　GEL 工具栏

（3）Project Toolbar

Project 工具栏提供了与工程和断点设置有关的命令。如图 4-22 所示，工程工具栏提供了以下命令。



图 4-22　Project 工具栏

第一个下拉列表：显示当前被激活的工程，列表中包含当前工作界面中所有工程。

第二个下拉列表：显示当前工程的编译、链接配置，列表中包含当前工程的所有已定义的配置选项。

- Compile File：编译文件；
- Incremental Build：对所有修改过的文件重新编译，再链接生成可执行程序；
- Build All：全部重新编译链接生成可执行程序；
- Stop Build：停止 Build 操作；
- Toggle Breakpoint：设置断点；
- Remove All Breakpoints：移去所有的断点；
- Toggle Probe Point：设置 Probe Point；
- Remove All Probe Points：移去所有的 Probe Point。

（4）Edit Toolbar

如图 4-23 所示，Edit 菜单提供了一些常用的编辑命令及书签命令。



图 4-23　Edit 工具栏

- Mark To：将光标放在括号前面，再单击此命令，则将标记此括号对内所有文本；
- Mark Next：查找下一个括号对，并标记其中的文本；
- Find Match：将光标放在括号前面，再单击此命令，则将光标跳至与之配对的括号处；
- Find Next Open：将光标跳至下一个括号处（左括号）；
- Outdent Marked Text：将所选择文本向左移一个 Tab 宽度；
- Indent Marked Text：将所选择文本向右移一个 Tab 宽度；
- Edit: Toggle Bookmark：设置一个标签；

- Edit: Next Bookmark：查找下一个标签；
- Edit: Previous Bookmark：查找上一个标签；
- Edit Bookmarks：打开标签对话框；
- Enable the external editor：使能外部编辑器，需要在 Option->Customize 菜单的 Editor Properties 属性栏中设置好外部编辑器的位置、参数等属性。

（5）Debug Toolbar

如图 4-24 所示，Debug 工具栏由 5 个子工具栏组成，从左至右，从上至下分别是 Context-Sensitive Stepping Toolbar、ASM/Source Stepping Toolbar、Target Control Toolbar、Debug Window Toolbar、Multiple Operations Toolbar，分别提供以下常用的调试命令：



图 4-24　Debug 工具栏

- Single Step：与 Debug 菜单中的 Step Into 命令一致，单步执行；
- Step Over：与 Debug 菜单中 Step Over 命令一致；
- Step Out：与 Debug 菜单中 Step Out 命令一致；
- Source-Single Step：与 Debug->Assembly/Source Stepping 菜单中 Source Step Into 命令一致；
- Source-Step Over：与 Debug->Assembly/Source Stepping 菜单中 Source Step Over 命令一致；
- Step Out：与 Debug 菜单中 Step Out 命令一致；
- Assembly-Single Step：与 Debug->Assembly/Source Stepping 菜单中 Assembly Step Into 命令一致；
- Assembly-Step Over：与 Debug->Assembly/Source Stepping 菜单中 Assembly Step Over 命令一致；
- Run to Cursor：运行到光标处；
- Set PC to Cursor：设置寄存器 PC 的值为鼠标所在的程序区地址值；
- Run：运行程序；
- Halt：中止程序运行；
- Animate：与 Debug 菜单中 Animate 命令一致；
- Register Window：观察或编辑 CPU 寄存器或外设寄存器值；
- View Memory：查看存储器指定地址的值；
- View Stack：查看堆栈值；
- View Disassembly：查看反汇编窗口；
- Multiple Operations Toolbar 下拉列表：选择多步操作的类型；
- Multiple Operations Toolbar 微调控件：选择多步操作的步数；
- Excute：执行多步操作。

（6）Plug-in Toolbar

如图 4-25 所示，Plug-in 工具栏由 2 个子工具栏组成，从左至右分别是 Watch Window

Toolbar、DSP/BIOS Toolbar，分别提供以下命令：



图 4-25　Plug-in 工具栏

- Watch Window：打开 Watch 窗口观察或修改变量；
- Quick Watch：打开 Quick Watch 窗口观察或修改变量，还可将方便地将变量加入 Watch 窗口；
- Open Message Log：与 DSP/BIOS 菜单中的 Message Log 命令一致，显示日志模块传送的信息；
- Open Statistics View：与 DSP/BIOS 菜单中的 Statistics View 命令一致；
- Open Host Channel Control：与 DSP/BIOS 菜单中的 Host Channel Control 命令一致；
- Open RTA Control Panel：与 DSP/BIOS 菜单中的 RTA Control Panel 命令一致；
- Open Execution Graph：与 DSP/BIOS 菜单中的 Execution Graph 命令一致；
- Open CPU Load Graph：与 DSP/BIOS 菜单中的 CPU Load Graph 命令一致；
- Open Kernel/Object View：与 DSP/BIOS 菜单中的 Kernel/Object View 命令一致。

4.3.3　CCS 工程开发实例

本节以一个简单的 DSP 程序为例，介绍使用 CCS 开发 DSP 应用程序的几个基本过程：创建、生成、调试和测试，同时还将介绍 Build 参数的设置、Watch 窗口、断点（Breakpoint）、探测点（Probe Point）、图表（Graph）及动画（Animate）的基本概念。

1．创建一个新工程

（1）从程序组或桌面上运行 CCS。如果在 CCS 设置程序中安装并设置了多个驱动程序，则运行 CCS 将启动 Parallel Debug Manager 的运行。此时用 4.3.2.2 节的方法启动 Simulator 的运行。如果身边有硬件仿真器及其驱动程序，也可直接启动 Emulator 的运行。

（2）在 Project 菜单中选择 New 项，将弹出 Project Creation 对话框。在 “Project Name”文本框内输入工程名：myvolume，此时第二个“Location”文本框中会同步在原始目录路径上自动添加 myvolume 目录，该目录在工程创建完毕后自动生成；单击文本框右侧按钮可选择预先生成的目录作为工程目录。本例的工程目录为：“C:\ti\myprojects\myvolume”。在 “Project Type”下拉列表中选择工程类型：“Excutable（.out）”或“Library（.lib）”，本例选择第一选项——可执行工程类型，如果需要生成库文件便于其他工程调用，可选择第二选项——库工程类型。最后在“Target”下拉列表中选择工程应用的芯片类型，本例中选择“TMS320C54XX”。设置如图 4-26 所示。

（3）单击 Finish 按钮新建工程完毕，CCS 界面如图 4-27 所示。工程文件名为 myvolume.pjt，此文件保存了工程的设置信息及工程中的文件引用情况。

2．创建一个新文件

CCS 中可以创建文本文件、.C 文件、.H 文件、DSP/BIOS 配置文件等。本工程应用中以创建.C 文件为例。

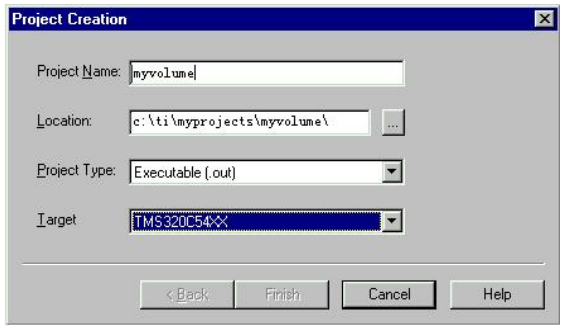


图 4-26　新建工程设置图

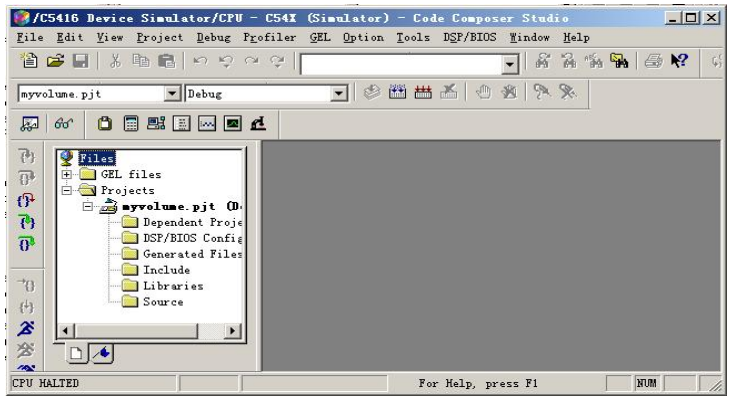


图 4-27　新建工程视图

（1）从 File 菜单中选择 New 子菜单，此时出现如下子菜单窗口，单击 Source File（或快捷键 Ctrl+N），出现标题为 Untitled1 的空白文件，如图 4-28 所示。

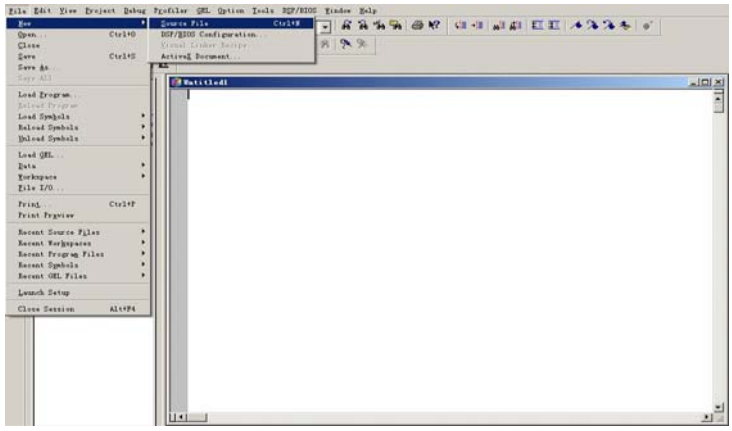


图 4-28　新建文件视图

（2）从 File 菜单中单击 Open（或快捷键 Ctrl+O）命令，弹出打开文件的对话框，在查找范围内选择 C:\ti\tutorial\sim54xx\volume1\volume.c 文件，选中该文件窗口，按 Ctrl+A 选择所有内容，单击 Edit 菜单中 Copy（或快捷键 Ctrl+C）命令复制所选内容。

（3）选择 Untitled1 文件窗口，鼠标单击第一行第一列，单击 Edit 菜单中 Paste（或快捷键

Ctrl+V）命令粘贴所复制的内容。此时窗口中出现的内容均为黑色，窗口标题栏 Untitled 后出现“*”符号，该符号表示文件被修改且没有保存。

（4）单击 File 菜单中 Save（或快捷键 Ctrl+S）命令，弹出保存文件对话框，选择工程目录：C:\ti\myprojects\myvolume，文件名设为：myvolume，保存类型设为：C Source File （*.c）。如图 4-29 所示。单击“保存”按钮保存文件，此时窗口标题为：myvolume.c，文件内容按照 C 语言的关键字、注释等不同类型用不同的颜色表示。

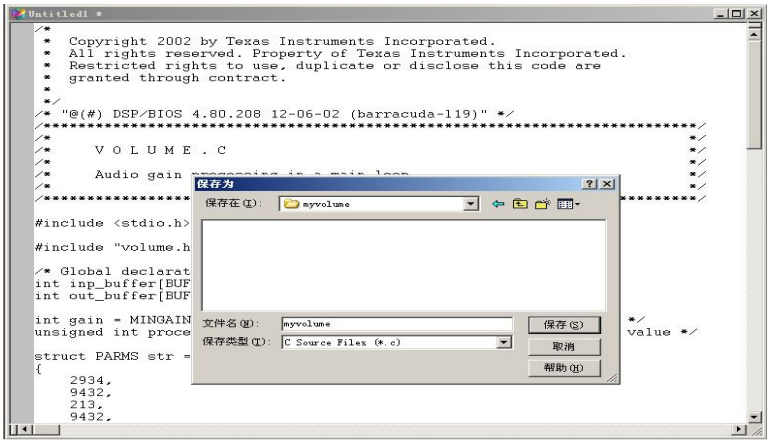


图 4-29　保存新建文件视图

（5）重复前 4 步操作，完成 volume.h、load.asm、vector.asm、volume.cmd、sine.dat 等文件的创建，或将 C:\ti\tutorial\sim54xx\volume1\目录下对应的文件复制到工程目录：C:\ti\myprojects\myvolume 下。

3．将文件添加入工程中

（1）从 Project 菜单中选取 Add Files to Project 命令，选择工程目录下 myvolume.c 文件，单击“打开”按钮将 myvolume.c 添加到工程中。

（2）将 vectors.asm 添加到工程中。vectors.asm 中定义了 DSP 的中断矢量，包含有不同中断的服务程序，如 RESET 中断服务程序为指向 C 程序入口 C_int00 函数。工程中也可用 DSP/BIOS 自动定义所有的中断矢量，DSP/BIOS 的相关知识请参考 4.3.4 节及 CCS 技术文档，这里不作详述。

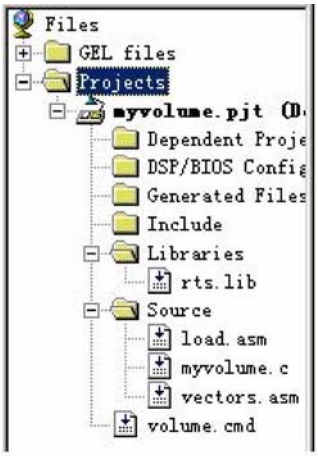


图 4-30　工程视图

（3）将 volume.cmd 添加入工程文件中，此文件的作用是将块（Sections）映射到存储器中。

（4）将 c:\ti\c5400\cgtools\lib 中的 rts.lib 加入到工程文件中，此文件为采用 C 开发 DSP 应用程序的运行支持库文件。

（5）在工程视图中双击所有“+”号，可看到整个工程的结构，如图 4-30 所示。如果没有看到工程视图，则需在 View 菜单下选择 Project 打开工程视图。还应单击工程视图左下角的 File 图标以确保能观察到图 4-30 所示的工程结构。

（6）在以上的操作中，没有将头文件加入到工程中，CCS 将在 Build 时自动查找所需的头文件。当 Build 完成时，可在 Project

视图中观察到生成程序所需的头文件。

如果需要从工程中去除一个文件，可将该文件选中，按 Del 键即可。也可单击右键，在弹出菜单中选择 Remove 将该文件从工程中移走。

当 Build 一个程序时，CCS 会自动依次从以下路径中查找工程需要的文件：

（1）包括源文件的目录；

（2）Project->Options 下 Complier 和 Assembler 选项的 Include Search Path 域中定义的路径；

（3）C54x_C_DIR（C 编译器）和 C54x_A_DIR（汇编器）环境变量中声明的路径，C54x_C_DIR 指向的路径中包括有含 rts.lib 文件的目录。

4．查看代码

双击 Project 视图中的 myvolume.c，将在右边的代码窗口中看到文件源代码。可以在 Option 菜单中选择字体大小。

```
#include <stdio.h>
#include "volume.h"
/* Global declarations */
int inp_buffer[ BUFSIZE];          /* processing data buffers */
int out_buffer[ BUFSIZE];
int gain = MINGAIN;                /* volume control variable */
unsigned int processingLoad = BASELOAD;   /* processing load */
struct PARMS str =
{
      2934,
      9432,
      213,
      9432,
      &str
};
/* Functions */
extern void load（unsigned int loadValue）;
static int processing（int *input, int *output）;
static void dataIO（void）;
/* ======== main ======== */
void main（）
{
int *input = &inp_buffer[ 0];
int *output = &out_buffer[ 0];
puts（"volume example started\ n"）;
/* loop forever */
while（TRUE）
{
/* Read input data using a Probe Point connected to a host file.
*    Write output data to a graph connected through a probe-point.
*/
dataIO（）;
```

```
#ifdef  FILEIO
puts（"begin prcessing"）                    /* deliberate syntax error */
#endif
/* apply gain */
processing（input, output）;
}
}
/* ======== processing ========
* FUNCTION: apply signal processing transform to input signal.
* PARAMETERS: address of input and output buffers.
* RETURN VALUE: TRUE. */
static int processing（int *input, int *output）
{
int size = BUFSIZE;
while（size--）{
*output++ = *input++ * gain;
}
/* additional processing load */
load（processingLoad）;
return（TRUE）;
}
/* ======== dataIO ======== *
* FUNCTION: read input signal and write output signal.
* PARAMETERS: none.
* RETURN VALUE: none. */
static void dataIO（）
{
/* do data I/O */
return;
}
```

从以上 C 代码可以看出，如果 FILEIO 未定义，此程序将只能在屏幕上显示“volume example started”信息。如果 FILEIO 已定义，程序还将在每次循环的开始显示“begin prcessing”信息。

5．生成和运行程序

（1）执行菜单命令 Project->Rebuild All 或在 Project 工具栏上单击 Rebuild All 图标，对工程重新编译、汇编和链接，Output 窗口将显示进行汇编、编译和链接的相关信息。

（2）执行菜单命令 File->Load Program，选择工程目录下 Debug 目录中 myvolume.out 并打开，将 Build 生成的程序加载到 Simulator 中。CCS 将自动打开一个反汇编窗口，显示加载程序的反汇编指令。同时主窗口下方的 Output 窗口将显示 stdout 输出。

（3）在反汇编窗口中单击汇编指令，按 F1 键将切换至在线帮助窗口，显示光标所在行的关键词的帮助信息。例如将光标放在 c_int00 行下的 STM 处，按 F1 将显示 STM 汇编指令的帮助信息。

（4）执行菜单命令 Option->Dis-Assembly Style Options，在弹出窗口中选择不同的选项，

查看反汇编窗口的变化。例如在 Dis-Assembly Style 下选择 Algebraic 并确认，则反汇编窗口将以代数指令显示反汇编代码。

（5）执行菜单命令 Debug->Run 或在 Debug 工具栏上单击 Run 按钮。由于运行支持库（rts.lib）中包括有 C 的输出 puts（）函数，因此可在 Output 窗口 Stdout 栏看到 volume example started 信息，表明程序已经运行，如图 4-31 所示。

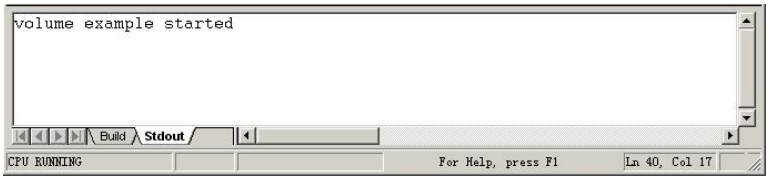


图 4-31　在 Output 窗口显示程序运行结果

6．更改Build选项并更正语法错误

由于以上程序中 FILEIO 没有定义，因此在编译时将忽略程序中的部分代码，在链接生成的 DSP 程序中也不包括这部分代码。下面通过更改程序选项来定义 FILEIO，从而将这部分代码生成到执行程序中，并更正源代码中存在的语法错误。

（1）执行菜单命令 Debug->Halt 或在 Debug 工具栏上单击 Halt 按钮，停止程序中死循环（while（TRUE））的运行，此时文件编辑窗口左侧出现黄色箭头标明程序停止的位置。

（2)执行菜单命令 Project->Build Options，弹出 Build Options for myvolume.pjt 窗口。

（3）单击 Compiler 栏，在 Category 域选择 Preprocessor，然后在右侧的 Define Symbols 域中输入 FILEIO，与_DEBUG 用分号隔开。此时单击编译参数栏，可以看到-d“FILEIO”，如图 4-32 所示。在定义 FILEIO 之后，C 编译器的编译范围将包括对应 FILEIO 预编译部分的源代码。单击“确定”按钮保存选项设置。

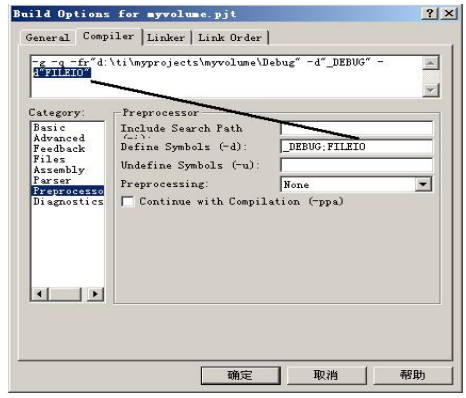


图 4-32　在 Build Options 下定义 FILEIO

（4）单击 Project 工具栏的 Rebuild All 按钮重新汇编、编译和链接。

（5）此时 Output 窗口 Build 栏将提示源代码中存在语法错误，错误出现在第 68 行，并提示该处需要一个分号（图 4-33）。根据 C 语言语法可知，上一行应该添加一个分号。在第 64 行 puts 函数后加上分号，保存文件，再单击 Project 工具栏的 Build 图标将修改后的文件重新编译并链接生成 DSP 程序。

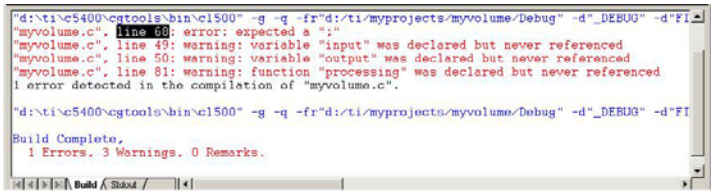


图 4-33　编译错误提示

7．使用断点调试程序

（1）执行菜单命令 File->Reload Program 重新加载程序。

（2）执行菜单命令 Debug->Go Main，此时文件编辑窗口左侧黄色箭头显示在 main 函数体开始处。移动光标，将其放在 puts（“begin processing”）；行上，并按 F9 或单击 Project 工具栏 Toggle Breakpoint 按钮设置断点，也可双击文件编辑窗口该行的左侧灰色部分设置断点，断点设置完毕后该位置出现红色圆点。执行菜单命令 Debug->Breakpoints，注意到在 Breakpoint 域可以设置一种 Break at Location if expression is TRUE 的断点，即根据一个表达式是否为真决定是否设置断点。此外还可以设置其他类型的断点。

（3）执行菜单命令 Debug->Run 或按 F5，运行程序。此时黄色箭头将停在断点处，Output 窗口 Stdout 栏显示上一条消息：“volume example started”。执行菜单命令 Debug->Step Over 或按 F10，单步执行程序，“begin processing”将出现 Output 窗口。在调试程序时，单步执行是非常有用的一种调试手段。Debug 工具栏中提供了两种不同单步模式：针对汇编语言和针对 C 语言，可以选择不同的命令，熟悉一下不同模式的调试方法。

（4）执行菜单命令 Debug->Animate 或按 F12，程序在断点处略作停顿则继续运行。Output 窗口不断出现“begin processing”。执行菜单命令 Debug->Halt 或按 shift+F5，停止程序运行。

8．使用Watch窗口观察变量

（1)执行菜单命令 View->Watch Window 或单击 Plug-in 子工具栏 Watch Window 中“Watch Window”按钮，在 CCS 主界面的右下方会出项变量观察窗口，如图 4-34 所示。观察窗口默认显示 Watch Locals 栏，该栏显示当前程序停止处的局部变量。

（2）单击观查窗口下方的 Watch1 栏，在“Name”列空白处输入“str”，回车或单击其该窗口空白处即可观察该变量的值。可以看到在 str 的左边有一个“+”标志，表明 str 是一个结构体。单击“+”将看到 str 结构体中包含的元素，双击每个元素可以更改其值大小，如图 4-35 所示。str 声明为 PARMS 结构体，在 volume.h 中定义。其中 Link 为一个指向结构体的指针，其地址值每次显示可能不同。单击 Link 左边的“+”号可以发现其无穷嵌套的特点。观察变量地址时，如果是数组元素，只需输入该数组名即可，如果是单个变量，则要在该变量名前加&符号。

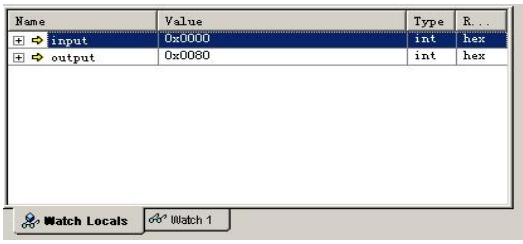


图 4-34　Watch Locals 窗口的局部变量显示

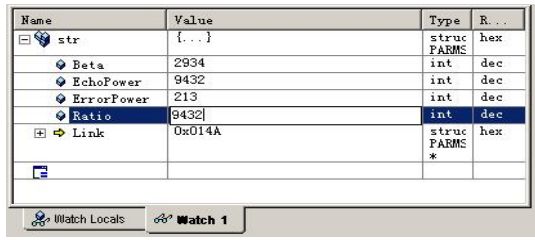


图 4-35　在 Watch1 窗口添加需观察的变量

（3）Watch1 窗口中选择一个变量，单击“Radix”列名，弹出如图 4-36 下拉菜单，通过该菜单可以改变所选变量的数据显示格式，如用十六进制（hex）显示、十进制显示（dec）等。

9．添加Probe Point从PC文件中读取数据

接下来介绍如何使用 Probe Point 从 PC 文件中存取数据。Probe Point 是开发算法的一个有用

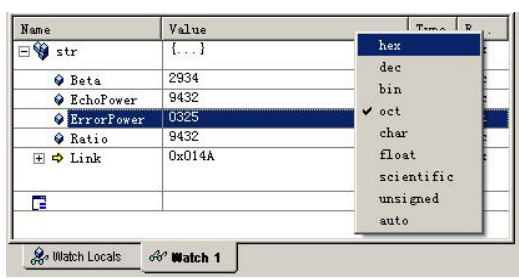


图 4-36　在 Watch1 窗口选择数据显示模式

工具。使用方法如下：

（1）将 PC 文件中数据传送给 Simulator 上特定数据区供算法使用；

（2）将 Simulator 上特定数据区中的输出数据传送到 PC 文件中以供分析；

（3）更新一个窗口，如由数据绘出的 Graph 窗口。

Probe Point 与 Breakpoint 都会中断程序的运行，但 Probe Point 与 Breakpoint 在以下几方面不同：

（1）Probe Point 只是暂时中断程序运行，当程序执行到 Probe Point 时会更新与之相连接的窗口，然后自动继续执行程序；

（2）Breakpoint 中断程序之后，将更新所有打开的窗口，且只能人工干预恢复程序运行；

（3）Probe Point 可与 FILEIO 配合，在目标板与 PC 文件之间传送数据，Breakpoint 则无此功能。

下面使用 Probe Point 将 PC 上文件送往 Simulator 上作为测试数据，同时使用断点在到达 Probe Point 时自动更新所有打开的窗口，这些窗口包括输入输出数据的图表。在操作前首先介绍一下 CCS 数据文件格式，本例中使用的数据文件 sine.dat 即为这种格式的文件。

CCS 允许的数据格式有两种，一种是 COFF 格式，即二进制的公共目标文件格式，可执行的 DSP 程序（*.out）即采用 COFF 格式；另一种是 CCS 数据文件。CCS 数据文件为字符格式文件，由文件头和数据两部分构成。文件头指明文件类型、数据类型、起始地址和长度等信息，相邻信息间以空格隔开。紧接文件头后面的为数据，每个数据占一行，数据类型可以为十六进制、整数、长整数和浮点数。

CCS 数据文件文件头格式为：

文件类型	数据类型	起始地址	数据页号	数据长度

（1）执行菜单命令 File->Reload Program，重新加载程序，然后执行菜单命令 Debug->Go Main，将程序停在 Main 函数体开始处。

（2）将光标放在主函数中 dataIO（）行上。

（3）在 Project 工具栏上单击 Toggle Probe Point 按钮或按 Ctrl+F9 添加 Probe Point。此时文件编辑窗口左侧灰色部分出现蓝色菱形小点，表明此处有一个 Probe Point。

（4）执行菜单命令 File->File I/O，出现 File I/O 对话框。可在对话框中选择输入输出文件。

（5）单击 File Input 栏，再单击 Add File 按钮。

（6）选择工程目录下 sine.dat 文件，文件类型选择*.dat（Hex）（默认情况），如图 4-37 所示。

（7）单击“打开”将文件添加到 File I/O 列表中。此时将出现一个控制窗口，如图 4-38 所示，可以在运行程序时使用这个窗口来控制数据文件的开始、停止、前进、后退等操作。

（8）在 File I/O 对话框中，在 Address 域填入 inp_buffer，在 Length 域填入 100，同时将 Wrap Around 框选中（图 4-39），这几部分值的含义如下：

- Address 域指示的是从文件中读取的数据将要放的地址，inp_buffer 是在 myvolume.c 中定义的整形数组，其长度为 BUFFSIZE；
- Length 域指示的是每次到达 Probe Point 时从数据文件中读取多少个样点，这里取值为 100 是因为 BUFFSIZE=100，即每次取 100 个样值存放在输入缓冲中，如果 Length 超过 100 则可能导致其他数据丢失；
- 选中 Wrap Around 表明读取数据的循环特性，每次在读至文件结尾处后将自动从文件头开始重新读取数据，这样将从数据文件中读取一个连续（周期性）的数据流。

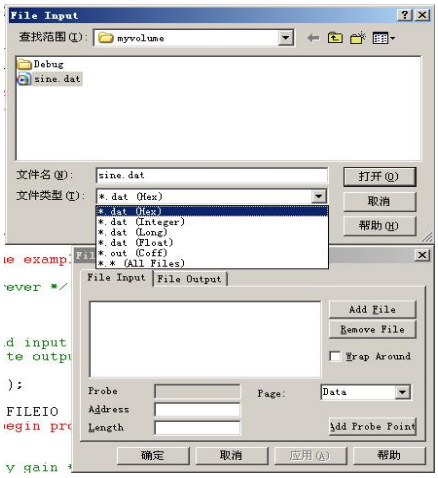


图 4-37　File I/O 控制窗口



图 4-38　File I/O 控制窗口

（9）单击 File I/O 对话框中的 Add Probepoint 按钮，将出现 Break/Probe Points 对话框并选中 Probe Points 栏。

（10）在 Probe Point 列表中，高亮显示写有 myvolume.c line 61→No Connection 行。表明第 61 行已设置 Probe Point，但还没有与 PC 文件相关联。

（11）在 Connect 域，单击向下箭头并从列表中选 sine.dat。

（12）单击 Replace 按钮，Probe Point 列表指示 Probe Point 已与 sine.dat 文件相关联。

（13）单击“确定”按钮，File I/O 对话框指示文件连至一个 Probe Point。

（14）单击“确定”按钮，关闭 File I/O 对话框。

10．利用图形功能观察数据

如果此时运行程序，将观察不到任何程序运行结果。当然也可以设置 Watch 窗口显示 inp_buffer 和 out_buffer 的值，但所要观察的变量实在太多，而且显示的是生硬的数字信息，与图形显示相比，既不友好也不直观。

CCS 提供很多方法将程序产生的数据画图显示，包括时域/频域波形显示、星座图、眼图及图像显示。下面使用频域/时域波形显示功能观察一个时域波形。

（1）执行菜单命令 View->Graph->Time/Frequency 弹出 Graph Property 对话框。

（2）如图 4-40 所示，在 Graph Property 对话框中，更改图形的标题、起始地址、采集缓冲区的大小、显示数据大小、DSP 数据类型、自动升缩属性及最大 Y 值。

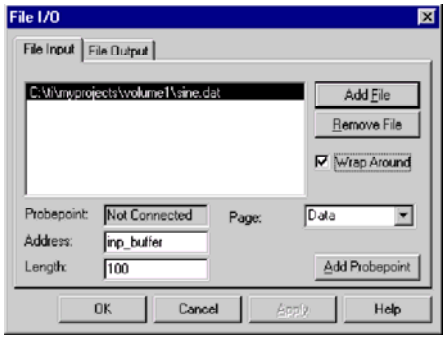


图 4-39　File I/O 属性

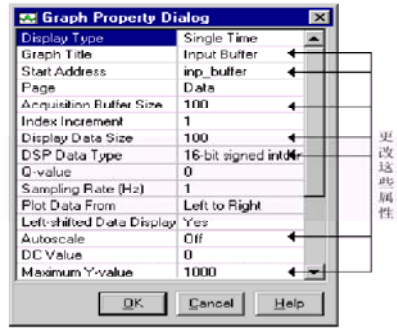


图 4-40　更改图形属性

（3）单击 OK 按钮，将出现一个显示 inp_buffer 波形的图形窗口。

（4）在图形窗口中击右键，从弹出菜单中选 Clear Display 清除已有显示波形。单击右键，选择弹出菜单 Float In Main Window，则窗口自动 CCS 主界面的右侧。此时单击右键，选择弹出菜单 Allow Docking，则窗口浮在主界面上。

（5）再次执行菜单命令 View->Graph->Time/Frequency。

（6）将 Graph Title 改为 Output Buffer，开始地址栏改为 out_buffer，其他设置不变。

（7）单击 OK 按钮，出现一个显示 Out_buffer 波形的图形窗口，同样击右键从弹出菜单中选 Clear Display 清除已有显示波形。

11．程序和图形的动画（Animate）显示

至此，我们已设置一个 Probe Point。它将临时中断程序运行，将 PC 上数据传给目标板，然后继续执行程序。然而 Probe Point 不会更新图形显示，因此我们将创建一个断点，使图形窗口自动更新。使用 Animate 命令，使到达断点并更新窗口后程序能自动继续执行。

（1）在 myvolume.c 窗口，将光标放在调用 dataIO 行上。

（2）在 Project 工具栏上单击 Toggle Breakpoint 图标设置断点。将断点和 Probe Point 设在同一行上可以使程序在只中断一次的情况下执行两个操作：传输数据和更新图形显示。

（3）重新组织窗口以便能同时看到两个图形窗口。

（4）在 Debug 工具栏单击 Animate 按钮或按 F12 键。此命令将运行程序，碰到断点后临时中断程序运行，更新窗口显示，然后继续执行程序。与 Run 不同的是，Animate 会继续执行程序直到碰到下一个断点。只有人为干预，程序才会真正停止运行。可将 Animate 命令理解为一个“运行→中断→继续”的操作。

（5）每次碰到 Probe Point 时，CCS 将从 sine.dat 文件中读取 100 个样值，并将它们写至输入缓冲 inp_buffer。由于 sine.dat 文件保存的是 40 个采样值的正弦波形数据，因此每个图包括 2.5 个 sine 周期波形，如图 4-41 所示。

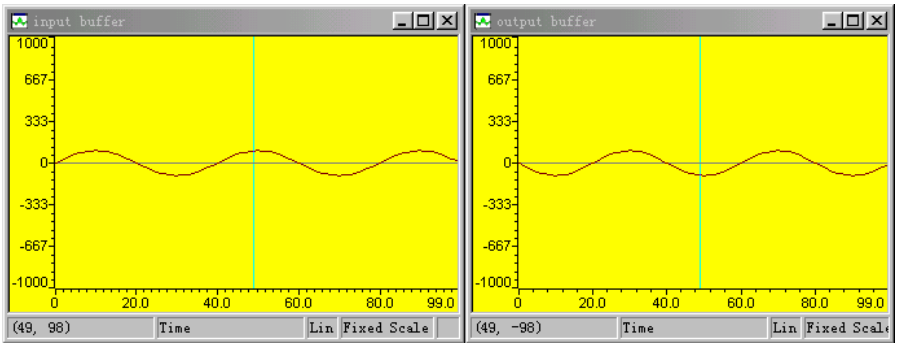


图 4-41　Gain=1 时的图形显示

从图 4-41 可以看出，input buffer 与 output buffer 的波形是反相的。这是因为 input buffer 中包含的数据是从 sine.dat 读来的，而 output buffer 的数据则是处理函数处理的最后一组值。使用图 4-38 所示的 File I/O 控制窗口可以控制数据文件的开始、停止、前进、后退等操作。在停止状态下，input buffer 与 output buffer 的波形应完全一致。

12．调节增益

本程序将输入缓冲的数据与增益相乘后送至输出缓冲中：

```
output++=input++*gain;
```

增益被初始化为 MINGAIN，在 volume.h 中定义为 1。为改变输出值，需改变增益，方法之一是使用观察（Watch）功能。

（1）执行菜单命令 View->Watch Window。

（2）按第 8 步观察变量操作，选择 Watch1 栏，输入 gain 作为要观察的变量。

（3）如程序已中止运行，单击 Animate 按钮重新运行程序。

（4）在 Watch1 栏双击 gain 变量对应的 Value 列，在编辑变量窗口将 gain 值改为 10，回车或单击其他空白处。

（5）注意到输出缓冲图中的幅度值已增大为原来的 10 倍，如图 4-42 所示。

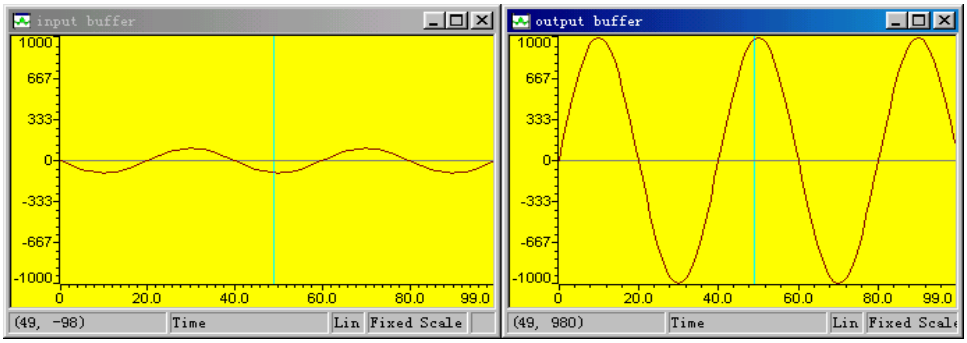


图 4-42　Gain=10 时的图形显示

13．查看可视范围之外的变量

以上使用 Watch 窗口观察并改变变量值。有时需查看不在现在断点可视范围内的变量，此时应使用 Call Stack 菜单命令。

（1）在 Debug 工具栏击 Halt 图标或按 Shift +F5 中断程序运行。

（2）查看 C 源代码，注意到*input 在 main 和 Processing 二函数中定义，在 dataIO 函数中没有定义。

（3）将光标放在 dataIO 函数中 return 行。

（4）在 Project 工具栏上击 Toggle Breakpoint 按钮或按 F9 键设置断点。

（5）按 F5 运行程序。由于该行不是一个有效行，因此 CCS 将在 Output 窗口 Message 栏给出提示（图 4-43），并自动将断点设在下一个有效行。

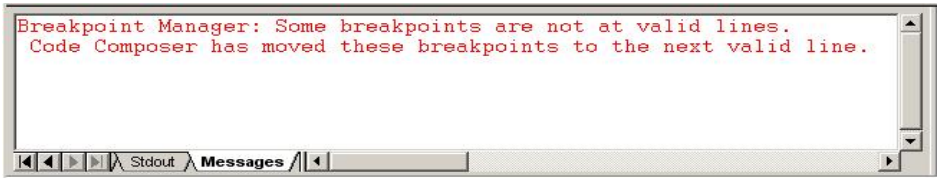


图 4-43　断点设在无效行的提示

（6）按 F5 运行程序。程序将运行到 dataIO 函数结尾处的断点时停止运行。

（7）在 Watch1 窗口输入*input，指示变量为一个未知的标识符，这是因为*input 在 dataIO 函数中没有定义。

（8）执行菜单命令 View->Call Stack，将在 Watch 窗口左侧看到 Call Stack 窗口。

（9）在 Call Stack 窗击 main()，观察到*input 在 main 函数的值为 0，如果读者更改 sine.dat 文件则其值可能改变。

（10）单击 Call Stack 窗的最后一行，可见 gain 为一个全局变量，但*input 不是。

（11）在 Call Stack 窗口中击右键，在弹出菜单中选 Hide。

（12）移去第 4 步中设置的断点（将光标放在 dataIO 函数中 Return 行后按 F9 键）。

4.3.4　DSP/BIOS

DSP/BIOS 是 TI 公司特别为其 TMS320C6000、TMS320C5000 和 TMS320C28x 系列 DSP 平台所设计开发的一个实时多任务操作系统内核。该操作系统由 TI 公司免费提供，本身占用很少的 CPU 资源，被集成在 CCS 开发环境里，其尺寸可裁剪。DSP / BIOS 提供底层的应用函数接口，可用于支持系统实时分析、线程管理、调度软件中断、周期函数、Idle 函数（后台运行函数）以及外部硬件中断与各种外设的管理。

1．DSP/BIOS配置

在 CCS 环境下，可静态创建和配置 DSP/BIOS API 使用的对象。此文件同时还可以简化存储器映射和硬件 ISR 向量映射。因此即使不调用 DSP/BIOS API 也可使用此配置文件。

配置文件有两个作用：

（1）设置全局运行参数；

（2）作为一个可视编辑器，创建和设置目标程序 DSP/BIOS API 调用的运行对象的属性，这些对象包括软件中断、I/O 管道和事件日志。

当打开一个配置文件时，会出现与如图 4-44 所示的窗口。

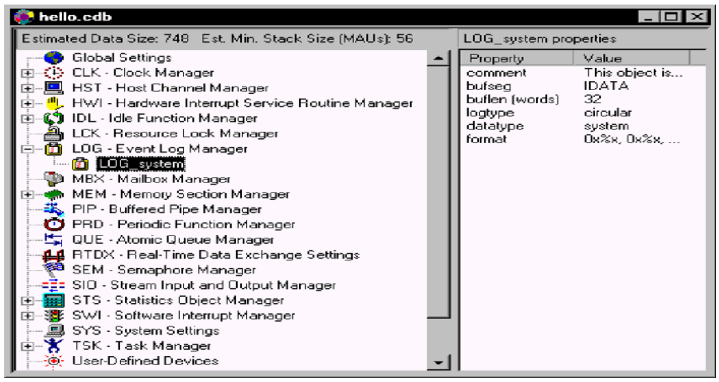


图 4-44　DSP/BIOS 配置

通用的 API 函数调用是在程序运行时创建对象的，这将导致在目标 DSP 占用额外的代码空间。与之不同的是，所有 DSP/BIOS 对象都是静态配置且在执行程序中占用固定的程序代码。这种静态配置特性还可以在程序执行之前通过验证对象属性的合法性来提前检测错误。

由于配置文件简化了程序的 ISR 矢量和存储器段寻址，因此它既可在调用 DSP/BIOS API 的程序中使用，也可在不调用 DSP/BIOS API 的程序中使用。

存储一个配置文件将同时产生一个链接命令文件（Programcfg.cmd）、一个头文件（Programcfg.h54）、一个汇编源文件（Programcfg.s54）、一个配置初始化文件（Programcfg_c.c）和对应的头文件（Programcfg.h），它们将自动加入到工程文件中。

2．DSP/BIOS API模块

在传统的调试手段中，调试处于执行程序之外。与传统调试不同，DSP/BIOS 应用程序要

求目标程序与特定的 DSP/BIOS API 模块链接起来。

通过在配置文件中定义 DSP/BIOS 对象，一个程序可以使用一个或多个 DSP/BIOS 模块。定义这些对象为外部函数，在源文件中就可调用 DSP/BIOS 函数。每个模块有自己独立的 C 头文件或汇编宏文件，这样在使用某个 DSP/BIOS 模块的时候不会将其他 DSP/BIOS 模块包含进来，从而减小程序代码量。

DSP/BIOS 提供了超过 150 个 API 函数，其中有与 C 语言、C++语言和汇编语言接口的函数。这些 API 函数是为在硬件目标的嵌入式程序工作的，包括实时统计、I/O 模块、软件中断管理和时钟管理等模块。由于应用程序的不同，DSP/BIOS API 函数的代码长度从 200~2000B 不等。DSP/BIOS API 调用（用 C 语言或汇编语言）时被优化以在目标 DSP 上使用最少资源。DSP/BIOS API 包含以下模块：

- ATM：提供原子函数以生成共享数据；
- C54：提供专用的 DSP 函数以管理中断；
- CLK：片内定时器模块，控制片内定时器并提供一个逻辑 32 位的实时时钟，其中断的最高速率与片内定时器寄存器一样（1 个指令周期），最低速率可以长达几个毫秒甚至更长；
- DEV：设备驱动模块，允许开发设计人员创建并使用自己的设备驱动；
- HST：主机输入输出模块，管理主机通道对象，使应用程序能在目标板和主机之间传输数据，主机通道被静态配置为输入或输出；
- HWI：硬件中断模块，为硬件中断例程提供支持，在配置文件中可指定硬件中断发生时要运行的函数；
- IDL：空闲函数模块，管理空闲状态函数，当目标程序没有更高优先级的程序执行时，DSP 处于空闲状态；
- LCK：锁定模块，管理共享的全局资源，当有多个任务竞争资源时仲裁资源的使用权；
- LOG：日志模块，管理 LOG 对象，在目标程序执行时实时捕获各种事件，可使用系统日志或定义自己的日志，可在 CCS 中实时观察这些日志；
- MBX：信箱模块，管理在任务间传递信息的对象，当等待信箱信息的时候任务被阻塞；
- MEM：存储器模块，为目标程序的不同代码或数据区指定存储段；
- PIP：数据管道模块，管理用来缓冲输入输出数据流的数据管道，这些数据管道提供一个一致的软件数据结构，用来在驱动 DSP 芯片与其他实时外设器件之间的输入输出；
- PRD：周期函数模块，管理触发程序中函数周期执行的周期对象，这些对象的执行速率可以由 CLK 模块的时钟速率控制，也可通过周期性调用 PRD_tick 来控制；
- QUE：队列模块，管理数据队列结构；
- RTDX：实时数据交换模块，使数据能在主机和目标板之间实时交换，然后在主机端分析和显示；
- SEM：信号量模块，管理任务之间同步和互斥操作的信号量；
- SIO：流模块管理，管理能提供高效、实时且与设备无关的输入输出的对象；
- STS：统计模块，管理当程序运行时存储关键统计数据的统计累加器，可以 CCS 环境下实时观察这些统计数字；
- SWI：软件中断模块，管理软件中断，当一个目标程序用 API 调用发送一个 SWI 对象时，SWI 模块调度相关函数的执行，软件中断共有 15 个优先级，但所有优先级都比硬件 ISR 优先级要低；

- SYS：系统服务模块，提供执行基本系统服务的通用函数，如中止程序执行和打印格式文本等；
- TRC：跟踪模块，管理一组跟踪控制位，通过事件日志和统计累加器实时捕获程序信息，由于不存在 TRC 对象，因此在配置文件中没有列出 TRC 模块；
- TSK：任务模块，管理比软件中断优先级还低的任务线程。

4.4　硬件开发

任何实用系统都是以硬件平台作为基础，硬件开发是整个系统开发的基础。DSP 芯片开发的硬件、软件开发可以同步进行，而接口部分软件开发与硬件开发更是紧密联系、不可分割的。

硬件开发流程如图 4-45 所示，首先按照功能需求分析选择合适的芯片，如针对 TI 的 DSP 芯片，一般的语音处理算法可选择 TMS320C54x、C55x 系列，图像处理算法则可选择 TMS320C64x 系列。选好后结合芯片本身的外围接口和系统功能需求，进一步选择外围接口芯片，如语音处理系统中常用的 A/D 芯片等。所有芯片选好后，按照其设计手册设计外围电路。绘制好系统原理图后，认真审核功能模块、接口模块、电源、复位、启动等系统模块。原理图正确无误后，可进行 PCB 布板，布板时要注意系统硬件尺寸、电磁兼容性等布线要求。PCB 审核后即可进行制板、焊接等硬件加工处理。焊接完毕的电路板首先要测试其电源模块，以防电源模块有问题，烧坏 DSP 芯片。按照功能划分，分别调试各个系统模块。最后结合底层软件接口模块、上层系统模块、算法模块，完成系统集成。

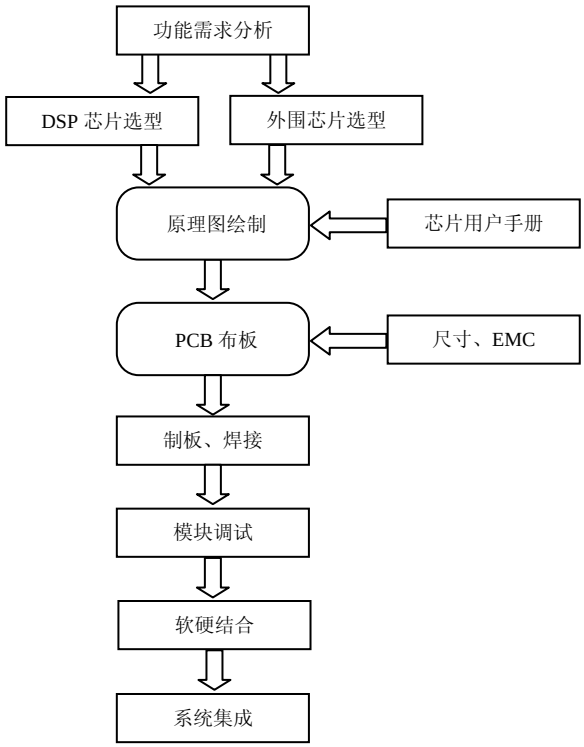


图 4-45　硬件开发流程

4.5 硬件开发环境

DSP 硬件开发通常基于 DSP 仿真器进行。DSP 仿真器，也即扩展开发系统（XDS），是进行 DSP 芯片硬件开发的最常用的工具，可分为电路型仿真器和扫描型仿真器两种。

早期的仿真器是电路型仿真器。采用电路型仿真器进行硬件仿真时，仿真器的电缆插头必须插入到用户硬件电路中 DSP 芯片的相应位置，也即是仿真电缆的插头引脚必须与 DSP 芯片的引脚一一对应。这类仿真器的主要缺点是，仿真时需要用仿真器上的 CPU 替代目标系统的 CPU、调试过程在监控程序控制下执行、需要占用目标系统软硬件资源等。TI 的 XDS/22 仿真器属于这一类，主要用于早期 TMS320C1x 和 TMS320C2x 等芯片的仿真。

现在普遍采用的是扫描型仿真器。现在常用的各型 DSP 芯片的仿真都采用这种仿真器。扫描型仿真器不采用插入仿真的方法，而是通过 DSP 芯片上提供的几个仿真引脚实现仿真功能。这种方法为 TI 公司所开发，可用来解决高速 DSP 芯片的仿真。由于高速 DSP 芯片具有高度并行的结构、快速的指令周期、高密度的封装等特点，采用传统的电路仿真方法很难实现可靠的仿真。扫描仿真消除了传统的电路仿真存在的问题，如仿真电缆过长会引起引起信号失真仿真插头会引起可靠性差等问题。用户程序可在目标系统的片内或片外存储器实时运行，而不会因仿真器引入额外的等待状态。

DSP 芯片内部是通过移位寄存器扫描链实现扫描仿真，这个扫描链被外部的串行口访问。采用扫描仿真，可以对已经焊接在电路板上的芯片进行仿真调试，这对于在生产过程中调试 DSP 系统也带来极大的方便。

目前 TI 公司的 DSP 芯片均提供有 JTAG（或 MPSD）仿真信号线。DSP 仿真器的一端提供一个 JTAG（或 MPSD）插头，与 DSP 芯片上提供的接口信号相连接，采用边界扫描的原理，访问 DSP 芯片的内部资源，并可控制 DSP 芯片访问其外围电路。仿真器的另一端可以通过 USB 总线、计算机并口、PCI 总线、ISA 总线与计算机连接。

目前，仿真器的仿真接口有 MPSD 和 JTAG 两种。

MPSD 接口：TMS320C3x（不包括 TMS320VC33）的仿真器采用了 12 线的仿真接口，仿真器通过这个仿真接口与目标系统的 TMS320C3x 相接进行仿真，其中，EMU0~EMU3 为四根仿真线，H3 为时钟线，由 TMS320C3x 提供，第 7 脚的 PD 为存在检测线，用于指示仿真头与目标系统的连接，在目标系统中，此脚接至电源。第 8 脚为空脚，其余均为地线。

JTAG 接口：其他 DSP 芯片的仿真器接口都采用 JTAG 标准，符合 IEEE 1149.1 技术规范。表 4-12 是 JTAG 仿真信号的定义，其中 TDO 信号与 TCK 时钟的下降沿对齐，TMS 和 TDI 在 TCK 时钟的上升沿取样。

采用 DSP 仿真器，可对 DSP 系统进行实时仿真。对用户来说，在设计 DSP 硬件系统时，需要根据仿真器接口信号的定义，在电路板上设计相应的仿真接口，图 4-46 给出了 JTAG 和 MPSD 仿真接口的定义。

JTAG 定义

TMS	1	2	$\overline{\text{TRST}}$
TDI	3	4	GND
PD（Vcc）	5	6	No pin（key）
TDO	7	8	GND
TCK_RET	9	10	GND
TCK	11	12	GND
EMU0	13	14	EMU1

MPSD 定义

EMU1	1	2	GND
EMU0	3	4	GND
EMU2	5	6	GND
PD（Vcc）	7	8	No pin（key）
EMU3	9	10	GND
H3	11	12	GND

图 4-46 JTAG 与 MPSD 仿真接口定义

表 4-12　JTAG 仿真信号的定义

仿真头信号	仿真头状态	DSP 芯片状态	信号说明
TMS	输出	输入	JTAG 测试方式选择
TDI	输出	输入	JTAG 测试数据输入
TDO	输入	输出	JTAG 测试数据输出
TCK	输出	输入	JTAG 测试时钟；TCK 是仿真电缆头输出的时钟源，该信号可用来驱动系统测试时钟
$\overline{\text{TRST}}$	输出	输入	JTAG 测试复位
EMU0	输入	输入/输出	仿真脚 0
EMU1	输入	输入/输出	仿真脚 1
PD	输入	输出	存在检测；该信号用于指示仿真电缆连接正确，目标系统已加电；PD 在目标系统中应接至电源
TCK_RET	输入	输出	JTAG 测试时钟返回；测试时钟输入至仿真头，该信号是缓冲或非缓冲的 TCK 信号

实际应用时，将 DSP 仿真器的两端分别与计算机和 DSP 系统相连接，运行 CCS 软件就可对目标系统进行在线仿真。

4.6　小　　结

本章分别介绍了 DSP 芯片开发的软件、硬件开发流程，重点讲解了软件开发环境。以 TI 公司 CCS 集成开发环境为例，重点介绍了 CCS 的安装、设置，对 CCS 菜单、工具栏进行了详细的说明。通过对一个 DSP 应用实例的分析，介绍了 DSP 软件开发中经常使用的各种方法和工具，如工程建立、编译参数设置、调试工具、变量观察、数据输入等，同时给出了 DSP/BIOS 的基本介绍。了解并熟练掌握这些开发工具对于加快开发速度提高开发效率十分重要。

4.7　习题与思考题

（1）DSP 芯片软件开发的工具分哪几类，各有什么作用？

（2）从 Sample.c 源程序到最终的 Sample.out，中间需要经过哪些步骤？

（3）CCS 集成开发软件有哪些功能，与原先的 DSP 开发软件相比有哪些优势？

（4）CCS 的 Simulator 和 Emulator 有何区别，在哪些情况下适合使用 Simulator 调试程序，哪些情况下必须使用 Emulator 调试程序？

（5）编写一段 FIR 滤波器的 DSP 程序，用 CCS 数据格式编写一个双音多频（DTMF）的计算机文件，按 4.3.3 节的方法测试 FIR 滤波器算法。

（6）什么是电路型仿真？什么是扫描型仿真？

（7）简要说明采用 DSP 仿真器如何进行硬件系统的仿真。

第 5 章　基于 C 语言的 DSP 芯片开发

5.1　引　　言

对于 DSP 软件开发而言，DSP 芯片和通用处理器（如 80x86）一样，都是通过指令执行的方式实现各种功能的。每种芯片结构不同，都有自己独特的指令系统和对应的机器操作码。由于汇编语言指令与芯片的结构紧密相关，因此采用汇编语言编写 DSP 程序，可以有效减少程序的运算量和存储量，具有代码效率比较高的优点，这对于运算资源和存储资源有限的 DSP 系统来说是十分有用的。然而，基于汇编语言开发 DSP 程序，由于需要掌握每一条汇编指令的用法，因此程序开发的周期相对较长，且编写的程序可读性和可移植性较差，软件的修改和升级相对也比较困难。

基于上述原因，DSP 芯片厂商都推出了基于高级语言的开发环境，以方便用户开发应用程序，其中以 C 语言为基础的开发环境最多。这样可以直接用高级语言编写 DSP 芯片的软件，从而使 DSP 芯片的开发速度大大提高，也使程序的修改和移植变得十分方便。当然，基于 C 语言开发 DSP 程序有一个前提条件，那就是所设计的 DSP 系统必须具有足够的运算和存储资源。

第 4 章已经介绍了 TI 公司的 DSP 集成开发环境 CCS。本章将在此基础上，重点讨论在 CCS 环境中如何利用 C 语言进行 DSP 程序的开发。直接利用 DSP 汇编语言指令进行开发的方法将在本书的第 6 到第 8 章介绍。

5.2　CCS 支持的 C 语言

目前常见的各种 DSP 开发软件，其 C 语言编译器基本上都是符合 ANSI C 标准的。第 4 章已经介绍，CCS 环境中集成了 DSP 代码的生成工具，其中包括了优化的 C/C++编译器、汇编器、链接器以及一系列相应的辅助工具等。这些工具程序都可以在 CCS 中直接完成。

下面以 C54x 的 C 编译器为例进行说明。利用 C 语言实现一个简单的累加程序 main。

```
/* main.c */
int func（int,int,int,int,int）;
int y = 0;
void main（void）
{
      y = func（1,2,3,4,5）;
}

/*     func.c     */
int func（int a,int b,int c,int d,int e）
{
```

```
    return（a+b+c+d+e）;
}
```

在 CCS 中，这样的程序可以直接进行编译，而不会有错误或告警产生。C5000 系列的 CCS C 编译器支持符合 ISO C 标准的 C 语言程序。这个标准可在 ISO 的规范和 Kernighan 与 Ritchie 合写的《C 编程语言》（第 2 版）中查阅。此外，C5000 系列的 CCS C 编译器还同样支持 C++语言，其版本是由 Ellis 和 Stroustrup 的《Annotated C++ Reference Manual》手册规定的。此外，C54x 的 C++编译器也支持 ISO 14882-1998 的嵌入式 C++标准。

为了保证程序运行结果的可靠，CCS 的 C 语言程序运行环境对 DSP 芯片中影响运算结果的各状态比特位中进行了缺省设置。表 5-1 列出了 C54x 芯片 C 语言环境中各状态比特位取值。

表 5-1　C54x 芯片 C 语言环境中各状态比特位取值

状态寄存器中的比特	名称	预置值	能否在 C 语言中修改
ARP	辅助寄存器指针	0	可以
ASM	ACC 移位模式		可以
BRAF	块重复激活标志		不可以
C	进位比特		可以
C16	双 16 位运算模式	0	不可以
CMPT	兼容模式	0	不可以
CPL#	编译模式	1	不可以
FRCT#	小数模式	0	不可以
OVA/B	累加器溢出标志		可以
OVM	溢出模式	0	*
SXM	符号扩展模式		可以
SMUL	饱和/乘法控制模式		*
TC	测试控制标志		可以

注#：如果程序员修改了预设值，这个值必须由函数恢复。

注*：只对指令有效。

为了方便用户使用 C 语言开发 DSP 程序，TI 等公司提供了运行支持库（run time support library）文件 rts.lib，其源码保存在 rts.src 文件中。这个文件包含了 ISO 定义的标准运行支持函数、编译工具函数、浮点代数运算函数和 C 语言 I/O 函数等。由于所使用的 DSP 平台有区别，用户有时需要定制库文件。利用库定制工具程序 mk500，按如下方法即可生成 rts.lib：

mk500 – –u –o2 rts.src –l rts.lib

其中，– –u 选项通知 mk500 工具使用当前路径下的头文件，–o2 是优化选项，对代码进行寄存器、局部和全局等优化。这个优化选项并不影响代码的兼容性。



5.2.1　变量和常数

5.2.1.1　基本数据类型

ANSI C 语言中的基本数据类型在 CCS 的 C 编译器中都可以直接使用。但由于芯片结构不同，有些数据类型的占用位宽有所区别。

在 C54x 和 C55x 中，整型数定义为累加器的低半部分。因此，整型数定义类型的占用位宽和具体的芯片类型有关，表 5-2 是 TI 公司主要 DSP 芯片的整型数宽度。

表 5-2　TI 公司主要 DSP 芯片整型数宽度

芯片类型	short （unsigned short）	int （unsigned int）	long （unsigned long）
C2x/C54x/C55x	16	16	32
C64x	16	32	40

TMS320C2x/C5x/C54x 的 C 编译器同样支持浮点型变量，它将浮点数表示为 IEEE 单精度格式。单精度数和双精度数都表示为 32 位，两者没有任何区别。

表 5-3 是 C54x 的 C 编译器支持的基本数据类型列表。从表中可以看出，除整型数和浮点数的格式与在 PC 上开发的 C 语言程序规定不同外，字符型、指针型等数据类型也存在差别。因此，在程序移植时需要考虑变量的变化范围。

表 5-3　C54x 的 C 编译器支持的基本数据类型

数据类型	位宽/bit	表示形式	最小值	最大值
signed char	16	ASCII	−32768	32767
char,unsigned char	16	ASCII	0	65535
short,signed short	16	二进制补码	−32768	32767
unsigned short	16	二进制	0	65535
int,signed int	16	二进制补码	−32768	32767
unsigned int	16	二进制	0	65535
long,signed long	32	二进制补码	−2 147 483 648	2 147 483 647
unsigned long	32	二进制	0	4 294 967 295
Enum	16	二进制补码	−32768	32767
Float	32	32 比特 IEEE 形式	1.175 494e−38	3.402 823 46e+38
Double	32	32 比特 IEEE 形式	1.175 494e−38	3.402 823 46e+38
long double	32	32 比特 IEEE 形式	1.175 494e−38	3.402 823 46e+38
指针	16	二进制	0	0x0FFFF

5.2.1.2　变量的存储

1．局部变量和全局变量

在函数内部定义的变量是内部变量，它只在本函数范围内有效，也只有在函数内才能使用它们，这些变量称为“局部变量”。即使是在主函数 main 中定义的变量，也只在主函数中有效，而不因为在主函数中定义就可以在整个程序中有效。

全局变量是在函数之外定义的变量，程序中使用的外部变量都是全局变量，其有效范围是从定义变量的位置开始到文件结束。

在选择使用局部变量和全局变量时，需要分析它们的特点和程序的应用场合。利用全局变量，可以增加函数间数据联系的通道，方便变量的修改和传递，减少内存空间以及传递数据时的时间消耗。然而，由于全局变量始终占用存储单元，因此如果使用过多全局变量，这些变量的空间将得不到释放，同时使得函数的通用性和程序的清晰性降低，程序的可靠性和通用性将下降。

2．动态存储变量和静态存储变量

局部变量和全局变量讨论的是变量的作用域。如果从变量的生存期上分析，可以分为静态存储变量和动态存储变量。顾名思义，这两种变量的存储空间分别是固定的和动态分配的。在 C 语言中，除了变量之外，函数也有存储类别这个属性。

一般而言，DSP 的数据存储空间可以分为静态存储区和动态存储区。其中，全局变量保存在静态存储区中，程序开始执行时分配空间，程序执行完毕释放。动态存储区保存有函数调用时的形式参数变量、现场保护和返回地址等，还有未加 static 说明的局部变量。这些数据，在函数调用开始时分配空间，函数结束时释放。

对数据和函数的存储类别具体划分，可以分为四种：自动的（auto），静态的（static），寄存器的（register），外部的（extern）。其中任何一个文件中定义的数据和函数，缺省类型都是 auto。如果数据或函数在另外的文件中定义，则在使用文件中该数据应申明是 extern。另外两种存储类别的数据规定如下：

（1）静态变量

在函数内部定义的局部变量，缺省的都是动态分配存储空间的。这样，每次在调用该函数时，局部变量都要重新赋值。而有些情况下，局部变量在函数运行之后还要保存，以便下次调用时使用。这时，可以利用关键字 static 对变量进行定义。这种静态变量的定义形式如下：

static type varid；

C54x 等 C 编译器对静态变量的处理采用和全局变量相同的方法，在整个程序执行期间该变量都有效。

（2）寄存器变量

C54x 等编译器可以使用寄存器变量。这种类型的数据利用寄存器保存，指令处理起来比较快捷，可以减少对内存的访问。这种变量的定义形式如下：

register type regid；

C54x 的编译器在一个函数里最多使用两个寄存器变量。编译器通常利用 AR1 和 AR6 表示第一个寄存器变量和第二个寄存器变量。因此，regid 可以是 AR1 或者是 AR6。AR1 和 AR6 这两个寄存器是普通的保存入口（save-on-entry）寄存器。它们的类型不能是浮点或是长整型。

寄存器变量通常在变量列表处或者是函数的第一块中声明。嵌套块中的寄存器声明将按照普通变量对待。

变量的地址放在寄存器中，可以简化访问过程。因此，16 位类型的数据都可以作为寄存器变量使用。然而，由于每个寄存器变量在建立时都需要大概四个周期，因此如果使用寄存器变量，只用在这个变量的使用次数大于两次才能达到简化的效果。

下面是一个实例：

```
register struct data_struct *AR6
#define data_point （AR6）
data_pointer->element;
data_pointer++;
```

其中，第 1 条语句定义了寄存器变量，第 2 条语句定义 data_point 即为 AR6，可用 data_point 代替 AR6 进行寻址，后面两条语句即利用 AR6 对其结构中的数据进行访问。

通常，使用寄存器变量有以下两个原因：

（1）可能有一个全局变量在整个程序中都使用，而且如果将这个变量恒定地指定为寄存器变量，将显著降低代码大小，提高计算速度。

（2）使用中断服务程序时，如果调用非常频繁，那么定义寄存器变量后每次调用的时候不再保护和释放那么多寄存器，将提高运算速度。

5.2.1.3 其他数据类型

1．常数类型

C 语言中支持很多关键字。其中，const 这个关键字对数据目标的保存提供了很大的控制。可以使用这个关键字确保变量或数组的值不会被改变。

关键字 const 在变量定义时的位置很重要。例如，下面的两个例子：

```
int * const p = &x;
const int * q = &x;
```

第一个例子定义了一个固定的整型指针 p，而第二个例子定义了一个指针 q，它指向固定的常数。

2．端口类型

C54x 器件的编程中还支持 ioport 关键字，用于端口类型变量的定义，使用的方法如下：

```
ioport type portnum
```

其中 **ioport** 是这个端口变量声明的关键字，**port**hex_num 表明了这个端口的序号，hex_num 是个十六进制的数，这个端口可以是常见类型定义中的 char、short、int 以及其他的无符号变量。

端口变量的定义必须在文件级声明，在函数级的声明并不支持，因此不能在函数中使用关键字 ioport。例如，如果要在端口 10h 上声明端口变量，并将 a 输出到该端口上，再读取数据保存到 b 中，则函数可以这样实现：

```
ioport unsigned port10;
int func （）
{
    …
    port10 = a; /*向端口 10h 写 a*/
    …
    b = port10; /*从端口 10h 读数到 b */
…
}
```

端口变量可以和其他变量一样在计算等语句中使用，例如：

```
a = port10 + b;          /* 从端口 10h 读数，加上 b 后再赋给 a */
port10 += a;             /* 从端口 10h 读数，加上 a 后再写到端口 10h 上 */
```

函数调用的时候，端口变量利用实值传递，不需要通过虚参（reference），例如下面的语句，第一个语句是正确的，而第二个语句是不正确的。

```
call（port10）;          /* 从端口 10h 读数并传递给函数 call */
call（&port10）;             /* 不合法的传递 */
```

3．volatile类型

在对 C 源程序进行编译时，代码优化器会对数据流进行分析，避免任何可能发生的存储区访问。如果在编写 C/C++源代码时，存在一些依赖于存储区访问的代码，需要使用 volatile 关键字来确定这些访问。这样编译器不会对这些 volatile 变量进行优化。

例如，在下面的例子中，循环始终保持在固定的位置，直到读到的数为 0xFF。

```
unsigned int *ctrl;
while （*ctrl !=0xFF）;
```

由于*ctrl 循环不变，因此会被优化为一个存储区读指令。为了避免这样的优化，需要将 ctrl 定义为 volatile：

```
volatile unsigned int *ctrl;
```

这样定义后，对 ctrl 的访问就不会随着编译选项的改变而改变，确保了程序按照最基本的功能执行。

5.2.1.4　可寻址寄存器的定义

在 DSP 等芯片的平台上编写程序，常常需要对芯片的工作状态进行设置。这时常常要直接去访问各种各样的寄存器。汇编语言中有对寄存器访问的指令，但标准 C 语言中并没有给出这些寄存器的具体访问方法。那么该用什么方法去处理呢？

其实，DSP 芯片片内的寄存器，绝大多数都是可寻址的。因此，可以采用指针的方式进行定义，通过地址寻址的方式访问寄存器。例如，中断屏蔽寄存器（IMR）的地址为 0，则可以利用宏定义如下：

```
#define    IMR              （unsigned int *）0x0000
```

为保证这样的指针定义不受编译选项的影响，在 reg.h 中采用的方法是将 DSP 寄存器地址定义为 volatile 整型指针类型。C54x 的 CPU 寄存器具体定义如下所示：

```
#define   IMR              （volatile unsigned int *）0x0000
#define   IFR              （volatile unsigned int *）0x0001
#define   ST0              （volatile unsigned int *）0x0006
#define   ST1              （volatile unsigned int *）0x0007
#define   AL               （volatile unsigned int *）0x0008
#define   AH               （volatile unsigned int *）0x0009
#define   AG               （volatile unsigned int *）0x000A
#define   BL               （volatile unsigned int *）0x000B
#define   BH               （volatile unsigned int *）0x000C
#define   BG               （volatile unsigned int *）0x000D
#define   T                （volatile unsigned int *）0x000E
#define   TRN              （volatile unsigned int *）0x000F
#define   AR0              （volatile unsigned int *）0x0010
#define   AR1              （volatile unsigned int *）0x0011
#define   AR2              （volatile unsigned int *）0x0012
#define   SP               （volatile unsigned int *）0x0018
#define   BK               （volatile unsigned int *）0x0019
#define   BRC              （volatile unsigned int *）0x001A
#define   RSA              （volatile unsigned int *）0x001B
#define   REA              （volatile unsigned int *）0x001C
#define   PMST             （volatile unsigned int *）0x001D
#define   XPC              （volatile unsigned int *）0x001E
```

在主程序中，若要读出或者写入一个特定的寄存器，就要对相应的指针进行操作。下例通

过指针操作对 SWWSR 和 BSCR 进行初始化。

```
#define   SWWSR              （volatile unsigned int *）0x0028
#define   BSCR               （volatile unsigned int *）0x0029
int func （ ）
{
...

*SWWSR = 0x2000;
*BSCR = 0x0000;
...
}
```

也可以直接定义指针类型变量实现 MMR 的访问，例如：

```
volatile unsigned int *SWWSR =  （volatile unsigned int *）0x28;
```

这样就可以直接赋值：

```
*SWWSR = 0x8244;
```

在 regs54xx.h 和 regs55xx.h 中包含了大多数的 MMR 定义，用户可以在文件中查看引用。



5.2.2 函数

在 CCS 中利用 C 语言进行编程，其程序结构符合 ANSI C 标准。任何函数都是由三种基本的程序结构组成：顺序结构、分支跳转结构和循环结构。在 DSP 平台上开发 C 语言程序，要结合 DSP 的特点，因此对函数的编译有其特殊的规定。

5.2.2.1 函数及其调用

CCS 中 C 语言程序的函数定义和声明符合 ANSI C 标准。最基本的定义如下：

```
type function（type var，……）
{
……
}
```

其中，括号内的参数是函数 function 所要使用的变量，函数的返回值其类型为 type，没有返回值定义其类型为 void。

同时，C54x 的 C/C++编译器支持关键字 near 和 far 来声明函数调用的方式。这两个关键字可以认为是保存类型。它们可以出现在保存类型声明之前、之后或是之间。例如：

```
far int foo（）;
static far int foo（）;
near foo（）;
```

当使用 near 时，编译器使用 CALL 指令调用该函数；使用 far 时，使用 FCALL 指令调用该函数。为了声明函数的调用方式，也可以使用-mf 编译选项来设置，使用该选项时实现远调用，不使用时实现近调用。

需要指出的是，使用 near 和 far 时函数的指针并不改变，依然是 16 位的。而使用-mf 选项

时，函数的指针变为 24 位，这样可以访问扩展存储空间。

5.2.2.2　C 语言程序入口函数

1．main函数的调用——入口函数c_int00

对于 DSP 芯片的完整软件系统而言，用户开发的 main 函数只是一个应用程序。那 main 函数的调用过程由谁负责呢？完成这个功能的函数代码在 boot.c 中实现，它保存在 rts.lib 中。因此，在编译时需要将 rts.lib 添加到项目中。

当使用 C 语言编写 DSP 程序时，芯片的复位中断程序缺省为 c_int00，这实际上就是 C/C++ 程序的入口位置。DSP 芯片上电后，其程序指针（PC）指向中断矢量表的基址处（有关中断矢量表的内容请见 9.2 节），并从该位置跳转到 c_int00 处。程序 c_int00 完成系统全局和静态变量的初始化、C 环境变量的初始化、堆栈的建立和函数 main 的调用，由于没有任何调用者，因此 c_int00 并不保护任何寄存器的内容。

读者在进行程序调试时，在向 DSP 装载程序后，可以在 CCS 的汇编窗口看到类似于下面一段代码的汇编指令。这实际上就是 C 语言程序入口函数 c_int00 的汇编指令实现。

```
0000:1807          c_int00
0000:1807 7718          STM     920h,18h                ;设置堆栈基址
0000:1809 6BF8          ADDM    3ffh,*（18h）
0000:180C 68F8          ANDM    0fffeh,*（18h）
0000:180F F7B8          SSBX    SXM                     ;设置寄存器
……
0000:184D F074          CALL    main                    ;调用 C 语言主函数 main
0000:184F F074          CALL    _exit                   ;调用 C 语言返回函数
```

为了能够让 DSP 芯片复位后从 c_int00 开始执行程序，需要在复位中断矢量处写如下代码：

```
.ref     c_int00
.sect    "vectors"
rsv:   B      c_int00
```

由于 DSP 的 C 语言程序在运行 c_int00 之后才能进入到 main 函数中，main 函数运行结束后就跳转到 c_exit 中，因此调试过程中，只能在 main 函数运行时观察到变量窗中各个变量值。而在 PC 上编写 C 程序时，一旦调试就直接进入 main 函数，而没有 c_int00 的这个执行过程。这一点需要读者仔细理解。

2．变量的初始化

c_int00 在完成系统的初始化时，除了对一些必要的寄存器进行设置之外，还对 C 语言应用程序中的静态和全局变量进行必要的初始化。初始化的过程通过将存储在固定数据区空间的数据搬移到程序变量使用的 RAM 区中实现。

C 语言标准规定，在程序运行前任何没有特殊初始化的静态和全局变量都要初始为 0。当程序装载后，这个任务一般都要完成的。然而代码的装载过程非常依赖于应用系统的特定环境。CCS 编译器对没有特殊初始化的静态和全部变量不进行初始化。这个初始化任务必须由用户编写的程序完成。

const 类型定义的静态和全局变量与其他的静态和全局变量存在不同，它们保存在缺省的单独定义存储块（.const 块）中。在初始化的过程中，也不会对 const 类型的常量进行搬移。

例如：

```
const int zero=0;
```

这条初始化赋值语句在编译后并没有指令与其对应，而是直接在数据存储空间中的某个存储单元保存初始化值。

使用 const 类型定义变量，可以在系统上电初始化时不再浪费时间或空间，这个特点对于大数据量的常数初始化而言是有用的。

5.2.2.3 中断函数

在实时系统中，一般都会遇到中断处理的情况。在 C 语言程序中，利用 interrupt 关键字定义函数，可以声明该函数是一个中断函数。中断函数的编译服从专门的寄存器保护规则和堆栈返回过程。当 C/C++代码被中断后，中断服务程序必须对该程序和其他程序的函数使用的所有寄存器进行保护。当使用 interrupt 关键字后，编译器将为该中断提供寄存器保护和返回处理。

interrupt 关键字只能用于没有调用参数和返回值的函数。中断函数体中可以有局部变量，例如：

```
interrupt void int_handler（）
{
    unsigned int flags;
    …
}
```

在 C5x 的 C 编译器中，为简化中断函数的定义，每个中断函数采用固定的程序名。如下所示：

```
c_int0    系统复位中断
c_int1    外部中断 0
c_int2    外部中断 1
c_int3    外部中断 2
c_int4    内部定时器中断
c_int5    串行口接收中断
c_int6    串行口发送中断
c_int7    TDM 口接收中断
c_int8    TDM 口发送中断
c_int9    外部中断 3
```

调用上述中断程序时，首先调用一个名为 I$$SAVE 的子程序，这个子程序保护了所有的寄存器。同样，在函数返回时，调用一个名为 I$$REST 的子程序用于恢复被保护的寄存器。

用 C 语言编写中断程序时，需要注意以下几点：

（1）对由 SP（AR1）指向的字，编译器可能正在使用，因此必须加以保护。

（2）中断的屏蔽和使能必须由程序员设置，设置的方法是用嵌入汇编语句的方法修改 IMR 寄存器。这样修改不会破坏 C 环境或 C 指针。

（3）中断程序没有参数传递，即使说明，也将被忽略。

（4）由于用 C 编写中断程序时，需要保护所有的寄存器，因此效率不高。

（5）将一个程序与某个中断关联时，必须在相应的中断矢量处放置一条跳转指令。实现的具体方法将在后面的章节中介绍。

5.2.3　预处理

TMS320C5000 的 C/C++编译器支持预处理功能。除了一般的预定义、条件编译等预处理指令外，还有一个指令（directive）#pragma 可以指定预处理器完成和函数的存储、编译选项有关的一些任务，在对程序进行链接时有用。C54x 的 C 编译器支持下列的 pragma 预处理。

```
#pragma CODE_SECTION（symbol,”section name”）[;]
//可以为 symbol 指定专门的代码存储空间，在代码目标链接过程中非常有用
#pragma DATA_SECTION（symbol,”section name”）[;]
//可以为 symbol 指定专门的数据存储空间，在数据目标链接过程中非常有用
#pragma FUNC_CANNOT_INLINE（func） [;]   //表明函数 func 不能被扩展为内联
#pragma IDNET （string） [;]                  //在目标代码中插入一段注释 string，
//但并不装载到处理器中
#pragma INTERRUPT （func） [;]            //表明 func 是中断函数
#pragma NO_INTERRUPT （func） [;]      //表明 func 无法被中断，如果中断服务程序不
//产生调用，这个程序将通过 RETF 指令返回
```

C++编译器支持的 pragma 格式和 C 编译器支持的格式有所区别，这里不再列写。由于函数名 func 和符号名 symbol 不能在函数体内定义或声明，因此 pragma 的定义必须放在函数 func 和符号 symbol 的任何声明、定义和参考之前。

5.3　C 语言程序代码的优化

在实时系统应用中，很多任务需要大量的计算资源和存储资源。而 DSP 芯片片内存储空间有限，CPU 的时钟也不可能无限制地提高。为了充分利用资源，需要在保证程序正确性的同时，尽可能地对程序代码进行优化。因此，DSP 的 C 编译器中包含了一个优化器，可以用于 C/C++程序的速度和空间优化。另外，在编写 C/C++程序时，也有一定的优化规则。按照这些规则编写程序代码，可以有效地控制资源的消耗。本节重点讨论 C 代码优化的多种方法。

5.3.1　C 语言程序代码编译分析

为充分了解 C 代码优化的基本原理，对编译器的编译过程进行分析是必要的。下面从表达式、变量的存储、函数的调用等方面分析编译器对 C 语言代码的编译方法。

5.3.1.1　表达式分析

在利用 C 语言编写 DSP 程序时，会大量运用计算表达式。如果程序中需要计算整型表达式，必须注意以下几点：

（1）算术上溢和下溢。即使采用 16 位操作数，TMS320C2x/C5x 也产生 32 位结果，而 TMS320C54x 则产生 40 位结果。因此，算术溢出是不能以一种可预测的方式进行处理的，需要结合使用的芯片类型具体分析。

（2）整除和取模。TMS320C2x/C5x/C54x 没有直接提供整除指令，因此，所有的整除运算“/”和取模运算“%”都需要调用库函数 div 来实现。

div 的调用格式为：div_t div（int numer, int denom）

其中 div_t 是一个预定义的结构，定义如下：

```
typedef struct
{
    int quot;          // 商
    int rem;           // 余数
}
```

（3）32 位表达式分析。下面的一些运算在函数调用时并不遵循标准的 C 调用规则：①变量的左移，②变量的右移，③除法，④取模，⑤乘法。这样做的目的在于提高程序运行速度和减少程序代码空间。

（4）C 代码访问 16 位乘法结果的高 16 位。采用如下方法可以访问 16 乘法结果的高 16 位，无须调用 32 位乘法的库函数。

- 有符号结果：

```
int     m1, m2, result;
result = （（long） m1 *（long） m2） >> 16;
```

- 无符号结果：

```
unsigned    m1, m2, result;
result =（（unsigned long）m1 *  （unsigned long）m2）>> 16;
```

TMS320C2x/C5x/C54x 的 C 编译器将浮点数表示为 IEEE 单精度格式。单精度数和双精度数都用 32 位表示。在 TMS320C2x/C5x 的浮点函数库中提供了一组浮点数学库函数，如加法、减法。乘法、除法、比较、整数和浮点数转换、标准的错误处理等。这些函数也不遵循标准的 C 调用规则。调用这些函数时，C 编译器先将参数压入运行堆栈，然后调用浮点库函数。函数执行时，首先将参数从堆栈中弹出，然后执行函数运算，最后将运算结果压入堆栈返回。

有些浮点库函数需要整型或长整型参数或返回整型或长整型结果，对这些函数，用累加器的低 16 位传递或返回整型数，而用累加器的所有 32 位传递或返回长整型数。

5.3.1.2 存储模式

TMS320C2x/C5x/C54x/C55x 定点处理器有两种类型的存储器：程序存储器和数据存储器。在程序存储器中主要包含可执行的程序代码，在数据存储器中，则主要包含外部变量、静态变量和系统堆栈。由 C 程序生成的每一块程序或数据都存放于存储空间的一个连续块中。关于程序或数据块的存储将在第 6 章中具体介绍，这里我们分析 C 系统堆栈的存储。

C/C++编译器使用系统堆栈来完成存储局部变量、向函数传递参数、保护处理器状态等功能。这个存储空间必须放置于单一的连续存储块中，其增长方向从高地址向低地址增长。不同的芯片，其堆栈管理使用的寄存器有所区别。

对于 TMS320C2x/C5x，定点 C 编译器采用下面两个寄存器来管理这个堆栈：

AR1——堆栈指针（SP），指向堆栈的顶部。

AR0——帧指针（FP），指向当前帧的开始。

在 TMS320C54x 中，有专门的 SP 寄存器管理堆栈。辅助寄存器 AR0~AR7 可直接用作指针或用于表达式中。在需要时，AR7 可用作帧指针。

每个函数运行时，都会在堆栈中建立一个新的帧，用于分配局部变量和临时变量。C 环境能够自动管理这些寄存器。

C 编译器的堆栈长度可由链接器确定，全局符号__STACK_SIZE 的值等于堆栈长度，单位为字节，缺省值为 1K 字节。同样，需要改变堆栈长度时，在链接时用 -stack 选项，并在其后指定一个数值。

5.3.1.3　存储空间分配

在 C 语言程序中说明的每一个外部或静态变量被分配给一个唯一的连续空间。空间的地址由链接器确定。编译器保证这些变量的空间分配在多个字中以使每个变量按字边界对准。

在对结构变量进行存储空间分配中，编译器为结构分配足够的字以包含所有的结构成员，在一组结构中，每个结构开始于字边界。而所有非域类型对准于字的边界。对域分配足够多的比特，相邻域组装进一个字的相邻比特，但不跨越两个字。如果一个域要跨越两个字，则整个域分配到下一个字中。

DSP 的 C 编译器同样支持动态存储空间分配。ANSI 标准建议设 4 个有关的动态存储分配的函数，即 calloc（）、malloc（）、free（）、realloc（）。实际上，许多 C 编译系统实现时，往往增加了一些其他函数。ANSI 标准建议在“stdlib.h”头文件中包含有关信息，但许多 C 编译要求用“malloc.h”，而不是“stdlib.h”。

ANSI 标准要求动态分配系统返回 void 指针，它可以指向任意类型的数据，但大多数 C 编译器提供的函数都返回 char 指针。因此，需要用强制类型转换的方法把 char 指针转换成所需类型。

5.3.1.4　寄存器规则

不论是浮点 C 编译器还是定点 C 编译器，都定义了严格的寄存器使用规则。这些规则对于合理运行 C 语言程序，提高 C 语言程序运行效率非常重要。

C 编译器使用寄存器的方法在使用和不使用优化器时是不一样的。因为优化器需要使用额外的寄存器作为寄存器变量以提高程序的运行效率。但函数调用时保护寄存器的规则在使用和不使用优化器时是一样的。

下面我们来介绍定点 C 编译器使用寄存器的规则。

对于 TMS320C2x/C5x，定点 C 环境中保留了三个寄存器：AR0、AR1、AR2，它们的作用如下：

- AR0－帧指针；
- AR1－堆栈指针；
- AR2－局部变量指针，用于计算局部变量的地址。

除了这几个寄存器外，DSP 芯片中还有其他一些辅助寄存器和各种状态寄存器。对于 TMS320C2x / C5x，还包括 T 寄存器和 P 寄存器。这些寄存器在使用时必须符合下列规则：

（1）辅助寄存器（ARP 和 AR0~AR7）

对于 TMS320C2x/C5x，函数进入和返回时，ARP 必须为 1，也就是说当前的辅助寄存器为 AR1。函数执行时可以是其他值。对于 TMS320C54x，ARP 必须为 0，即当前辅助寄存器为 AR0。

对于 TMS320C2x/C5x，AR0 和 AR1 可以在函数执行过程中修改，但它们必须恢复。AR2、AR3、AR4 和 AR5 可以自由使用，也就是说，在函数执行过程中可以修改，也不必恢复。

对于 TMS320C54x，AR0~AR7 都可以自由使用。

如果 AR6 和 AR7 用作寄存器变量。如果在函数中被修改，必须加以保护和恢复。

（2）状态寄存器

在 C 编译器中，始终假定 PM 状态位为 0。如果函数改变 PM 值，则在函数返回时必须将 PM 重新设置为 0。在 TMS320C5x 中，TRM 位必须保持为 0，硬件复位时 TRM 为 0。对其他状态位既可以修改，也不必恢复，如 DP、C、FSM、HM、INTM、OV、OVM、SXM、TC、TXM、CNF、FO 和 XF 等。

（3）其他寄存器

累加器 ACC 可以自由使用，不必保护和恢复。ACC 可以用来返回整数、指针和浮点值。对于 TMS320C2x/C5x，P 和 T 寄存器也可以自由使用。

5.3.1.5 函数调用

函数调用涉及参数传递、内存使用等问题。DSP 的 C 编译器规定了一组严格的函数调用规则。除了特殊的运行支持函数外，任何调用 C 语言函数或被 C 语言函数所调用的函数都必须遵循这些规则，否则就会破坏 C 环境，造成不可预测的后果。

1．参数传递

将参数传递给一个 C 语言函数时，必须遵循下列规则：

（1）函数调用前，将参数压入运行堆栈。

（2）以逆序传递参数。也就是说。第一个参数（最左边）最后压栈，而最后一个参数（最右边）最先压栈。

（3）若参数是浮点数或长整型数，则低位字先压栈，高位字后压栈。

（4）对于 TMS320C54x，调用函数时，第一个参数放入累加器 A 中进行传递。

（5）传递结构时，采用多字方式。对于 TMS320C54x，若参数中有结构形式，则调用函数给结构分配空间，其地址通过累加器 A 传递给被调用函数。

2．局部帧的产生

函数被调用时，编译器在运行栈中建立一个帧以存储信息。当前函数帧称为局部帧。C 语言环境利用局部帧来保护调用者的有关信息、传递参数和生成局部变量。每调用一个函数，就建立一个新的帧。

对于 TMS320C2x/C5x，寄存器 AR1 作为堆栈指针（SP）使用，AR0 作为局部帧指针（FP）使用，SP 指向栈顶，FP 指向局部帧。编译器在建立局部帧时完成如下工作：

（1）从 TMS320C2x/C5x 的内部堆栈中弹出返回地址，并压入 C 运行堆栈。

（2）将旧的 FP 的内容压入 C 运行堆栈，并将 FP 设置为当前的 SP。

（3）增加 SP。增加的值等于需要保存的局部变量的字的个数加 1，其中，额外的一个字位于帧的开始，用于存储临时变量。

（4）若函数使用 AR1，AR6 和 AR7 作为寄存器变量，则将它们的内容压入堆栈，然后装入相应局部变量的地址。

对于 TMS320C54x 来说，由于寄存器的规则有所不同，上面的局部帧的产生过程在具体的寄存器操作上相应有一些改变。在处理时，采用如下方法：

（1）调用函数首先将第一个（最左边）的参数置于累加器 A 中。

（2）调用函数按逆序将剩下的参数搬移到参数块中（即左边的参数放在低地址单元），这些参数块存放于堆栈的顶部。

（3）调用函数开始调用过程。

（4）被调用的函数通过调整堆栈指针的方法为局部变量和参数块分配存储空间。

空间的大小按如下公式计算：

局部变量的存储空间＋max＋padding

其中 max 是调用过程中参数块存放时需要使用的空间大小。padding 有可能为 0，或为 1，它是为了保证 SP 位于偶地址上而设置的。

3．函数结束

函数结束返回时，必须完成如下工作以恢复 C 调用环境：

（1）处理要传递给调用者的返回值。

（2）如果使用了 AR6 和 AR7，则必须予以恢复；对于 TMS320C54x，若使用了 AR1，也一样需要恢复。

（3）撤销为局部变量和临存值分配的空间。

（4）恢复原来的帧指针。

（5）将返回地址压入堆栈并返回调用程序。

需要注意的是：

（1）函数在 ACC 中返回函数值。整数和指针在 ACC 的低 16 位中返回，浮点数和长整型数使用 ACC 全部 32 位返回。

（2）由于用 ACC 返回函数值，因此，必须保证 ACC 不被结束程序所修改。

（3）参数不是由被调用的函数弹出堆栈，而是必须由调用函数弹出。因此，调用者可以传递任意数目的参数至函数，同时，函数也不必知道有多少个参数需要传递。

（4）对于 TMS320C2x/C5x，从函数返回时，ARP 指向 AR1；对于 TMS320C54x，ARP 指向 AR0。

下面是 C54x DSP 程序中被调用函数的 C 语言代码和汇编代码实例。经过比较可以看到，为了完成 C 语言函数调用，必须按照严格的规则执行相应的汇编代码。

【例 5.1】 C 语言函数和汇编语言函数的对应比较

```
void callee（int a, int b）
{
      int c;
      int d;
……
return;
}

callee:                          ;函数的入口地址
```

```
PSHM    AR6             ;保存 AR6
PSHM    AR7             ;保存 AR7
FRAME   #-15            ;利用 FRAME 修改 SP，用于存放局部帧和参数块
                        ; 函数主体
FRAME   #15             ;清除局部帧和参数块空间
POPM    AR7             ;释放 AR7
POPM    AR6             ;释放 AR6
RET                     ;返回
```

5.3.2 C 语言程序的优化方法

C 语言并不是专门为数字信号处理而设计的编程语言，因此要实现高效的信号处理，必须对 C 语言程序的代码进行优化。根据优化方法不同，可以分为算法优化、函数优化和编译优化。函数优化的内容由于涉及汇编语言，因此将在第 8 章中介绍。

5.3.2.1 算法优化

算法优化是指对程序要完成的功能合理选择算法及其实现方法。这是代码优化中最重要和最关键的一步。同样的任务，一般有不同的算法可以实现，这些算法性能有差异，计算量也有区别。只有选择合适的算法，才能高效地实现系统设计要求。

例如，在信号处理中最常用的谱分析，现在一般都采用快速傅里叶变换来实现。之所以选择这样的方法，正是多年来的研究和应用表明，这种方法计算量较小，性能较好，非常适合于在实时系统中使用。

关于算法的研究和实现不是本书的重点，这里不再详细介绍。

5.3.2.2 编译优化

在使用 C 语言编译器对源程序进行编译的时候，为了提高代码的运行速度，减少代码的占用空间，编译器可以根据程序员的要求对程序进行适当的优化，例如将循环语句简化、在寄存器中保存变量等等。使用优化 ANSI C 编译器，可以在 C 源程序级对程序进行调试，大大缩短 DSP 应用程序的开发周期。

由于不同芯片平台上的汇编源码不同，针对不同芯片的编译器，采用的优化方法有所区别。不论何种 C 编译器的优化，在设计时都需要考虑三方面的效率：

（1）产生可与手工编写相比的汇编语言程序；

（2）提供简单的 C 语言运行环境的程序接口，使得关键的 DSP 算法可用汇编语言实现；

（3）为用 C 语言开发高性能 DSP 应用，建立一定规模且使用方便的工具库。

DSP 的 C 编译器中提供了一个独立的优化编译器。采用优化编译可以生成效率更高的汇编代码，从而提高程序的运行速度，减少目标代码的长度。从某种程度上说，C 编译器的效率主要取决于 C 编译器所能进行优化的范围和数量。

由于优化器是独立的，在 C 语言编译的过程中可以激活，也可以不激活。优化器处于 C 编译器其他两个模块的中间，即在文法分析之后，而在代码产生之前，如图 5-1 所示。

激活优化器的方法有两种，一种是直接运行优化程序。这种方法实际上将整个 C 编译过程分为三步。第一步是运行文法分析程序，进行预处理工作，输入文件为 C 源文件（.C），输

出为中间文件（IF）；第二步就是运行优化程序，输入文件是.IF 文件，输出是.OPT 文件；第三步是运行代码产生程序，输入为.OPT 文件，输出是.ASM 文件。显然，用这种方法进行 C 编译不太方便。



图 5-1　C 编译器组成

比较常用的方法是所谓的一步编译汇编链接法。在 C 编译器中提供了一个外壳程序（Shell），采用这个程序可以使 C 程序的编译、汇编和链接三个步骤一次完成。采用不同的选项可控制编译、汇编和链接过程。当然优化器也是用相应的优化选项来激活的。

C 编译器的优化方法可以分为两类，一类是 C 语言的通用优化；另一类是根据 DSP 芯片的特定优化。对于 C 语言的通用优化，优化的范围主要有：

（1）简化表达式；
（2）数据流优化；
（3）删除公共子表达式和冗余分配；
（4）优化跳转；
（5）简化控制流；
（6）优化与循环有关的变量；
（7）将循环体内计算值不变的表达式移至循环体前；
（8）运行支持库函数的行内扩展。

对于 DSP 芯片的特定优化，优化范围主要有：

（1）高效地使用寄存器；
（2）自动增量寄存器寻址方式；
（3）使用块重复；
（4）使用并行指令；
（5）使用延迟跳转。

优化器的激活比较方便，只要在编译命令中添加-On（-on）选项即可，其中 n 表示了优化的水平，优化级越高，优化的范围就越广。TI C54x 系列 DSP 的优化器提供了 4 种不同水平的优化功能，具体如表 5-4 所示。在 CCS 中，可以在编译选项中选择，单击菜单 project 下的 build option 选项，会出现如图 5-2 所示的窗口，在其中可以看到 opt level 滚动栏，根据需要可以选择合适的优化选项。

-x 选项是一个次选项，这个选项用于行函数的扩展，当程序需要调用函数时，优化器可将函数直接嵌入程序的调用处，这样可以避免函数调用所需的调用返回等附加操作，从而加速程序的运行，当然这种优化是以增加程序代码长度为代价的。

-x 选项分为三种，其中 -x0 不进行行函数扩展，-x1 是缺省选项，即没有-x 选项时实际上就是 -x1 选项，-x1 选项进行固有操作的行函数扩展，如 abs、labs 和 fabs，选择 -x2 选项时，编译器建立 _INLINE 预处理器符号并赋值为 1，同时激活最高级的优化（-o2）。

表 5-4　C54x 的优化选项

优化层次	优化级别	优化处理
0 级优化（-O0）	寄存器优化	1．对控制流图进行简化 2．利用寄存器保存变量 3．对循环进行简化 4．分析代码，清除没有使用的代码 5．对表达式进行简化
1 级优化（-O1）	局部优化	除了寄存器优化外，还执行以下处理 1．进行单一函数的优化（局部变量/常数传递） 2．清除没有使用的赋值和通用表达式
2 级优化（-O2，-O）	全局优化	除了局部优化外，还执行以下处理 1．对循环进行优化 2．消除全局通用的子表达式 3．消除全局没有使用的赋值
3 级优化（-O3）	文件级优化	除了全局优化外，还执行以下处理 1．消除所有没有使用的函数 2．对带有返回值但确没有使用的函数进行简化 3．重新对函数的声明进行排序，这样当调用者被优化的时候，被调用函数的属性可以保留 4．对文件级的变量特性进行确认；利用项目模式进行多个文件的优化

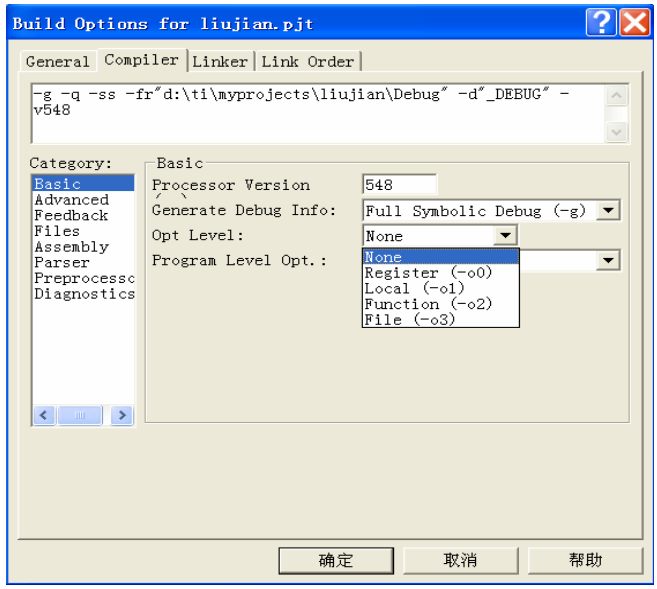


图 5-2　CCS 中 C 语言程序的优化选项

采用 C 优化编译可以提高程序的运行效率，加快程序的运行速度，但由于优化时采用了一些优化措施，使得 C 和汇编的交叉列表文件不如在不用优化时的那样清晰。此外，在调试程序时，最好先不用优化编译进行调试，待程序调试成功后再用优化编译进行验证。

采用 C 优化编译时，为了保证程序的正确性，特别需要注意以下几点：

（1）使用 volatile 变量避免优化。

（2）寄存器的使用。使用优化编译时，寄存器的使用比较灵活，更多的变量将存于寄存器中，虽然这不会影响正常 C 代码的正确性，但却会使一些假设变量在特定寄存器中的 asm 语句失效。此外，在中断程序中，必须将所有 C 函数可能用到的寄存器保存起来。

总之，使用 C 优化编译可以提高程序的效率，但同时也需十分谨慎。

下面我们举例说明 C 优化方法及其效果。

【例 5.2】高效地使用寄存器（TMS320C3x）

```
int gvar;
reg（int i, int j）
{
        gvar=call（）&i;
        j=gvar+i;
}
TMS320C3x C  编译器输出:
*
*R4  分配给变量  'i'
*R5  分配给变量  'j'
*     ...
        CALL    _call           ;R0=call（）
        AND     R4,R0           ;R0&=i
        STI     R0,@_gvar       ;gvar=R0
        ADDI    R4,R0,R5        ;gvar  在  R0 中,结果 j 存在 R5 中
```

例 5.2 是 TMS320C3x C 优化器高效使用寄存器的例子。在这个例子中，由于充分地使用了寄存器存储变量、参数和临存值，从而使优化编译得到的汇编程序的效率与用手工编写汇编程序的效率完全相同。此外，优化编译还可以以基于代价的方式分配寄存器，也就是说， 编译器可以根据用户变量和编译器临存值的类型和使用频率分配寄存器。循环体内的变量优先级高，使用不重叠的变量可分配给同一个寄存器。具有特定要求的变量可以固定分配到寄存器，如编译器将循环计数器分配到 RC（TMS320C3x）。

【例 5.3】循环体重复

```
add3（int a[3]）
{
        int i, sum=0;
        for（i=0;i<3;++i）
        sum+=a[i];
        return sum;
}
TMS320C3x C  编译器输出:
_add3:
        ...
        LDI         *-FP（2）,AR4  ;AR4=$a[0];
        LDI         *AR4++,RC      ;sum+=a[0]
        ADDI        *AR4++,RC      ;sum+=a[1]
        ADDI        *AR4++,RC      ;sum+=a[2]
        LDI         RC,R0          ;返回  sum
```

例 5.3 是一个循环体重复的例子。当编译器确定一个循环体较短且循环次数较少时，编译器就将循环体重复以避免循环操作，这也就避免了任何跳转或使用重复寄存器。

【例 5.4】行内函数的扩展

```
#include <stdlib.h>
struct s int a,b,c[10]; s;
struct t int x,y,z[10]; t;
proc_str（struct s *ps, struct t *pt）
{
        ...
        memcpy（ps,pt,sizeof（*ps））;
        ...
}
TMS320C3x C 编译器输出:
_proc_str:
*AR4 分配给变量 'memcpy_1_from'
*AR5 分配给变量 'memcpy_1_to'
*AR6 分配给变量 'ps'
*AR7 分配给变量 'pt'
*    ...
        LDI     AR7,AR4         ;memcpy_1_from=pt
        LDI     AR6,AR5         ;memcpy_1_to=ps
        LDI     *AR4++,R0
        RPTS    10
        STI     R0,*AR5++       ;扩展 memcpy:
        ||LDI   *AR4++,R0
        STI     R0,*AR5++
```

例 5.4 是运行支持库函数的行内扩展的例子。使用行函数扩展，可以节约与函数调用相关的开销，同时也为使用其他优化方法提供了更多的机会。

上面三个例子都是关于 TMS320C3x 的特定优化的和 C 语言通用优化的例子。实际上，C 编译器的优化范围很广，其优化性能还需要在进一步了解汇编语言后才能更全面的分析，限于篇幅，这里就不再作更多的介绍。

5.4 小　　结

本章我们介绍了用 C 语言开发 DSP 芯片的方法。用 C 语言开发 DSP 芯片缩短了开发周期，提高了程序开发的效率，也使程序的可读性和可移植性大大提高，对于系统的改进和升级换代也带来了极大的便利。当然，用目前的 C 编译器生成的程序代码，其效率还不能完全与手工编写的效率相比拟，因此实际 DSP 应用系统中往往采用 C 和汇编的混合编写方法，有关这一方面的内容我们将在后面的章节中进行介绍。

5.5 习题与思考题

（1）从 C 语言源程序到汇编语言程序，C 语言编译器完成何种处理？

（2）C 语言程序代码优化有几种方法？

（3）利用 C 语言实现中断函数，有什么要求？

第 6 章　DSP 芯片的存储资源管理

6.1　引　　言

前面已经介绍，DSP 芯片一般采用哈佛结构，这种结构将程序区、数据区和 I/O 区分开，每个存储区独立编址。为了提高运算速度，DSP 芯片内部提供了一定数量的存储器，用于存放程序和数据。本章主要介绍 DSP 芯片的存储器资源、存储区组织、DSP 芯片的代码存储结构、目标文件格式以及存储区的分配方法等内容。

6.2　DSP 芯片的存储器

存储器主要用于存储程序、数据、变量等。从存储器的位置来区分，分为片内存储器和片外存储器两类。其中，片内存储器是 DSP 芯片本身所固有的，而片外存储器是根据 DSP 系统需要在 DSP 芯片外部扩展的。

6.2.1　片内存储器

片内存储器由 DSP 芯片内部所提供，主要有 ROM、RAM 和 Flash 等类型。不同的 DSP 芯片所提供的片内存储器的类型和数量不同，如 TMS320VC5402 片内有 4 K 字的 ROM 和 16 K 字的 RAM，TMS320VC5409 片内有 16 K 字的 ROM 和 32 K 字的 RAM，而 TMS320VC5416 片内有 16 K 字的 ROM 和 128 K 字的 RAM，TMS320LF2407A 片内有 32 K 的 Flash 和 2 K 的 RAM。用户可根据需要进行选择。

1．ROM

通用 DSP 芯片中的 ROM 一般都包括引导装载（Bootloader）程序和中断矢量表。有些芯片（如 TMS320C54x）还有若干常用的数据表格，如正弦函数表、语音 PCM 编码用的 μ/A 律扩展数据表等）。有些芯片内部有较大空间的 ROM，ROM 空间的一部分还可以映射到数据空间和程序空间，可用于用户程序的掩膜。此时的 ROM 称为定制 ROM。用户可将测试通过的最终程序和数据以目标文件的格式提交给芯片生产公司，由公司将其转化为相应的代码并编程写入 ROM，一旦写入 ROM，则不能更改。

2．RAM

DSP 芯片中的 RAM 有 SARAM 和 DARAM 两种。

SARAM 称为单访问 RAM，即每个 SARAM 在 1 个机器周期内只能被访问 1 次，也就是说，每个机器周期只能进行 1 次读或写操作。

DARAM 称为双访问存储器，每个 DARAM 在 1 个机器周期内能被访问两次，因此在同一个周期内，CPU 和外设（如 BSP 和 HPI）可以对 DARAM 进行读和写操作。

RAM 可作为数据区或程序区使用，有些 RAM 空间还可以同时映射到程序和数据空间。

3．Flash

有些 DSP 芯片提供有较大空间的 Flash 存储器（如 TMS320F280x/F281x、TMS320LF240x），便于用户直接将程序和固定数据写入到 Flash 中运行，无须再在外部扩展存储器。片内 Flash 存储器被映射到程序存储空间。对于没有外部存储器扩展接口的器件，Flash 在器件中始终处于使能状态。但是对于有外部存储器扩展接口的芯片，复位时将根据 MP/$\overline{\text{MC}}$引脚的状态决定访问片内的 Flash 还是从片外存储器启动执行。

一般带有 Flash 的 DSP 芯片都提供加密功能，加密后，外部不能读出 Flash 中内容，这对于保护知识产权非常有用。Flash 存储器可以进行多次擦除和烧写，DSP 为每个 Flash 都提供一组控制寄存器来控制该 Flash 块的擦除、编程和测试，方便用户修改和升级程序。

6.2.2 外部扩展存储器

片内没有 Flash 的 DSP 芯片，一般都提供外部扩展总线，便于用户扩展外部存储空间。外部扩展存储器常用于存放用户程序。

用户程序一般在加电开机后由 DSP 芯片内部 ROM 中的 Bootloader 将存放在外部存储器中的用户程序引导加载到片内 RAM 中高速运行。如果片内 RAM 空间不够大，也可扩展外部 RAM。如果仅用于存放用户程序，由于还要 Bootloader 引导加载，因此存储器数据宽度可以是 8 位或 16 位；而如果用户程序需要在外部存储器中运行，则扩展存储器的数据宽度需与指令宽度相同（如定点芯片的 16 位、浮点芯片的 32 位）。

需要注意的是，即使外部存储器的速度足够快，也比不上程序在片内的运行速度。因此，系统设计时应尽可能选择片内 RAM 较大的 DSP 芯片，并将程序全部放在片内运行。

6.3 存储区的组织

DSP 芯片一般都具有 64 K 字的程序空间、64 K 字的数据空间和 64 K 字的 I/O 空间。有些芯片（如 TMS320C54x）还具有多达 256 K 字~8 M 字的扩展程序地址空间。在设计 DSP 系统时，必须要考虑存储器空间的组织。

DSP 芯片的存储器结构可以通过重叠和分页方法加以改变，从而调整不同类型的空间大小；同时，DSP 的总线结构也使得系统可以扩展存储器空间。

在 TMS320C54x 中，处理器模式状态寄存器（PMST，片内地址为 0x1D）的 3 个比特位（MP/$\overline{\text{MC}}$、OVLY 和 DROM）都影响存储器的结构。用户可以根据需要，通过修改 PMST 的比特设置，方便地配置存储空间。

本节主要以 TMS320VC5416 为例来说明。

6.3.1 程序空间

程序存储器主要用于存储程序指令。

DSP 芯片一般具有两种工作模式：微计算机模式和微处理器模式。工作模式是由 PMST 寄存器中的 MP/$\overline{\text{MC}}$ 位决定。若 MP/$\overline{\text{MC}}$=0，DSP 芯片工作于微计算机模式；若 MP/$\overline{\text{MC}}$=1，

DSP 芯片工作于微处理器模式。一般可通过设置 MP/$\overline{\text{MC}}$ 引脚的逻辑电平来设置工作模式，也可通过软件修改 PMST 的 MP/$\overline{\text{MC}}$ 位来改变。

微计算机模式：片内 ROM 映射到程序空间。复位后程序跳转至固定起始地址（如 TMS320VC5416 的 FF80H），再根据存放在片内的中断矢量表跳转至片内存放引导程序的地址，开始运行片内引导程序。

微处理器模式：片内 ROM 被禁止。复位后程序跳至固定起始地址，再根据存放在片外存储器的中断矢量表跳转至相应地址开始运行用户程序。

片内 RAM 可以通过设置 OVLY 位来控制映射区域。若 OVLY=0，没有片内 RAM 映射到程序空间，这时片内 RAM 作为数据存储器使用；若 OVLY=1，片内 RAM 同时被映射到程序区和数据区。

在微处理器模式下，该芯片程序区的 8000h~FFFFh 映射到外部。在微计算机模式下，C000h~FFFFh 在片内，其中 C000h~FEFFh 为片内 ROM；FF00h~FF7Fh 保留，用做芯片测试，FF80h~FFFFh 为片内中断向量表，用于存放复位、中断和陷阱向量等。在程序区的上半区（0000h~7FFFh），存储器结构根据 OVLY 的设置而不同。若 OVLY=0，上半区为外部；若 OVLY=1，上半区为片内的保留区和 DARAM，既能作为程序存储器又能作为数据存储器使用。

该芯片采用页扩展方法扩展程序空间，可以访问多达 8 M 字的程序空间，其地址线有 23 根，并用扩展计数寄存器（XPC）进行扩展程序空间寻址。同时，提供 FB、FBACC、FCALA、FCALL、FRET 和 FRETE 等特殊指令，配合进行扩展程序区的寻址。

该芯片有 64 K 字的片内 DARAM 和 64 K 字的片内 SARAM。其中，DARAM 的地址范围为 0080h~7FFFh 和 18000h~1FFFFh，SARAM 位于程序区的 28000h~2FFFFh 和 38000h~3FFFFh。需要注意的是，如果程序存储器使用了片内 RAM（OVLY=1），访问程序空间时，不管 XPC 值为多少，访问地址 xx0000h~xx7FFFh 都被映射到 0000h~7FFFh 的片内 RAM。



6.3.2　数据空间

DSP 芯片一般具有 64 K 字的数据寻址空间。片内的 DARAM 和 SARAM 均可映射到数据存储空间。此外，也可将片内 ROM 的一部分映射到数据空间。方法是设置 PMST 寄存器中的 DROM 比特位：若 DROM=0，片内 ROM 不映射到数据空间；若 DROM=1，部分片内 ROM 映射到数据空间，并且当 MP/$\overline{\text{MC}}$=0 时，片内 ROM 同时映射到数据空间和程序空间。每一次复位时，DSP 都将 DROM 位清零。

TMS320C54x 的片内 RAM 划分为 8 块，将 RAM 以块来组织可以提高系统的性能。因为对于 DARAM 来说，如果两个操作数都存放在相同的块中，则访问需要 2 个周期；但若两个操作数分别存放在不同的块中时，访问仅需要 1 个周期。因此，用户可以设计数据读取区域，在 1 个指令周期内，从一个 DARAM 块中预取 2 个操作数，然后再写到另一块 DARAM 中。合理组织存储器可以提高 DSP 的运行效率。

在数据区的 60h~7Fh 单元，有一个 32 字的 DARAM 块，称之为便笺本 RAM 块。该块可以进行各种类型的高速暂存，避免将一个大的 RAM 块分割开。

需要注意的是，如果采用从外部存储器引导装载程序方式，必须将外部存储器的地址空间映射到数据区。

6.3.3 I/O 空间

TMS320C54x 提供了 I/O 空间用于对片外设备的访问，其地址空间共 64K 字（0000h~FFFFh）。采用 $\overline{\text{IOSTRB}}$、R/$\overline{\text{W}}$、$\overline{\text{IS}}$ 等信号可以实现对外部 I/O 设备的访问。两条特殊指令 PORTR 和 PORTW 用于对 I/O 空间数据的存取。

通常，I/O 设备的存取速度比 DSP 芯片的存取速度要慢。DSP 芯片提供了外部 READY 信号和软件等待状态产生机制来实现与外部慢速设备之间的通信。

HOLD 模式可以允许外设控制 DSP 芯片的外部总线，获取外部程序、数据和 I/O 存储器空间的资源。当 DSP 芯片处于 HOLD 状态（可通过设置 DSP 芯片的 HOLD 引脚实现）时，DSP 芯片的外部总线处于高阻状态，此时外设可完全获取对 DSP 芯片外部资源的控制权。

6.3.4 存储器映射寄存器

在 DSP 数据空间的前 60h 个地址单元内，包含有 CPU 和片内外设的映射寄存器。通过访问映射寄存器就可以实现对 CPU 和外设的相应操作。例如，在函数调用或中断时，可以完成寄存器内容的保存以及返回时寄存器内容的恢复；还可以完成累加器和其他寄存器之间的信息传送。使用映射寄存器简化了 CPU 和片内外设存储器的访问方式。

（1）CPU 寄存器。总共有 26 个寄存器，见表 6-1，访问这些寄存器时没有等待状态。

（2）外设寄存器。驻留在数据空间的 20 h~50 h，可以作为外设电路的数据寄存器或控制寄存器。外设寄存器存在于一个专用的外设总线结构中，它可以从外设中接收数据或者发送数据至外设总线。设置或清除控制寄存器的比特位可以激活、屏蔽或者重新配置外设状态。不同类型的 TMS320C54x 芯片具有不同的外设寄存器，但如果芯片具有某种外设，则控制该外设的外设寄存器就一定存在。

表 6-1 CPU 映射寄存器

地址（Hex）	寄存器	说　明	地址（Hex）	寄存器	说　明
0	IMR	中断屏蔽寄存器	E	T	T 暂存器
1	IFR	中断标志寄存器	F	TRN	状态转移寄存器
2~5	—	保留	10~17	AR0~AR7	辅助寄存器 0~7
6	ST0	状态寄存器 0	18	SP	堆栈指针寄存器
7	ST1	状态寄存器 1	19	BK	循环缓冲区大小寄存器
8	AL	ACCA 低位字	1A	BRC	块重复计数寄存器
9	AH	ACCA 高位字	1B	RSA	块重复起点寄存器
A	AG	ACCA 保护位	1C	REA	块重复终点寄存器
B	BL	ACCB 低位字	1D	PMST	处理器模式状态寄存器
C	BH	ACCB 高位字	1E	XPC	程序计数器扩展寄存器
D	BG	ACCB 保护位	1E~1F	-	保留

6.3.5 TMS320VC5416 的存储资源

TMS320VC5416 的存储资源包括：8M 字的程序寻址空间、64K 字的数据寻址空间和 64K 字的 I/O 空间；128K 字的片内 RAM，其中有 64K 字的 DARAM 和 64K 字的 SARAM；16K 字的片内 ROM。图 6-1 是该芯片存储资源结构示意图。

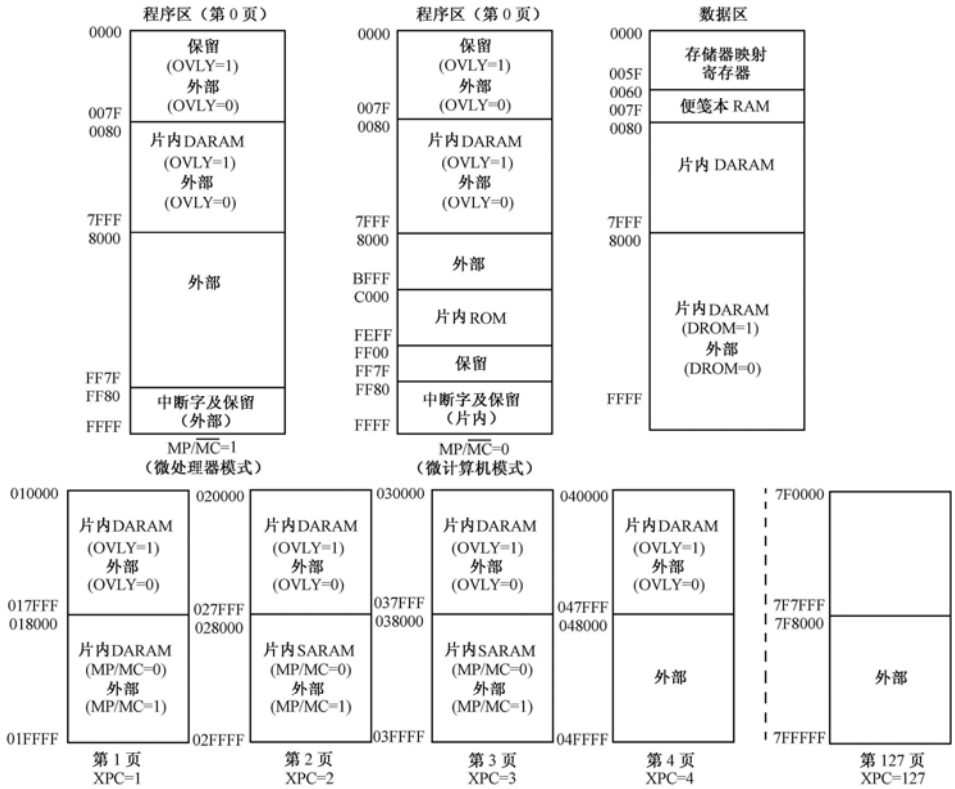


图 6-1　TMS320VC5416 存储资源结构示意图

64 K 字的 DARAM 分为 8 块，每块 8K 字，包括 DARAM0~7。其中，DARAM0~3 可作为数据存储器，地址范围为 80h~7FFFh，也可同时作为程序/数据存储器（OVLY=1），地址范围不变。DARAM4~7 作为程序存储器时，地址范围为 18000h~1FFFFh；将 DROM 设置为 1，还可作为数据存储器，地址范围为 8000h~FFFFh。必须注意，DARAM4~7 作为程序/数据存储器时，虽然地址不一样，但物理上是重叠的，如数据的 8000h 与程序的 18000h 实际上是同一个存储单元。

64 K 字的 SARAM 也分为 8 块，每块 8K 字，包括 SARAM0~7。这 64 K 字都位于程序空间，其中 SARAM0~3 的地址范围为 28000h~2FFFFh，SARAM4~7 的地址范围为 38000h~3FFFFh。

16 K 字的 ROM 位于程序空间的 C000h~FFFFh。表 6-2 列出了 ROM 的内容。

表 6-2　TMS320VC5416 的 ROM 内容

地址范围	内　容
C000h~D4FFh	GSM EFR 声码器数据
D500h-F7FFh	保留 可用于用户程序掩膜
F800h-FBFFh	Bootloader 程序
FC00h-FCFFh	μ 律扩展数据表
FD00h-FDFFh	A 律扩展数据表
FE00h-FEFFh	正弦数据表
FF00h-FF7Fh	芯片测试用
FF80h-FFFFh	中断矢量表

6.4 DSP 芯片中的代码存储结构

DSP 芯片的程序代码一般采用模块化结构存储。模块化结构的特点是结构清晰，便于程序代码的管理、分配与移植。为了实现这种结构，TI 公司的编译器和链接器所创建的目标文件采用一种称为 COFF 的目标文件格式（Common Object File Format，COFF）。采用 COFF 目标文件格式，有利于模块化编程，并且为管理程序代码和系统存储空间提供灵活的方法。基于 COFF 格式编写汇编程序或 C 程序时，不必为程序代码或变量指定目标地址，这为程序编写、程序移植、程序升级提供了极大的方便。

COFF 文件格式的核心是使程序员在编写 DSP 程序时基于代码块和数据块的概念，而不是一条条指令或一个个数据。基于这种文件格式编写的 DSP 程序具有可读性强、可移植性好的特点。在 COFF 文件格式中，将代码块和数据块称为 section，编译器/汇编器和链接器都提供了有关的命令来创建块和对块进行处理，编译器/汇编器创建的块的名称有些是已经定义好的，程序员也可自己定义块名。

本节着重对 COFF 文件格式中的块进行介绍，主要介绍块的基本概念、汇编器和链接器处理块的方法及程序的重定位方法等。

6.4.1 块

目标文件中的最小单位称为块（section），一个块就是最终在存储器映像中占据连续空间的一块代码或数据。目标文件中的每一个块都是相互独立的。一般地，COFF 目标文件应包含三个默认的块：

- .text 块 通常包含可执行代码；
- .data 块 通常包含已初始化的变量；
- .bss 块 通常为未初始化的变量保留空间。

此外，汇编器和链接器允许编程者建立和链接自定义的块，这些块与 .text、.data 和 .bss 相类似。所有的块可以分为两大类，即已初始化块和未初始化块。已初始化块包含程序代码和数据，.text 和 .data 及用 .sect 和 .asect 汇编器命令所创建的块都属于这一类。未初始化块是为未初始化的数据在存储器映像中保留空间，.bss 和用 .usect 汇编器命令创建的块属于这一类。汇编器提供了若干将各种代码和数据段与相应的块联系的命令，汇编器是在汇编过程中建立这些块的。图 6-2 给出了一个包含 .text 、.data 和 .bss 块的目标文件。

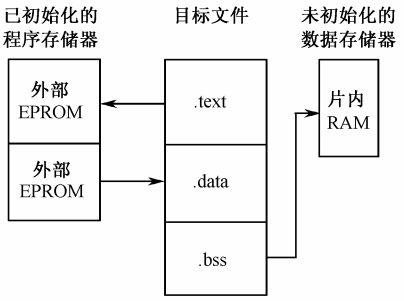


图 6-2 将存储器划分为逻辑块

链接器的一个主要功能是将块定位至目标存储器中。在汇编器汇编形成的目标文件中，除了用.asect 命令定义的块之外，其他程序块或数据块的地址都是不确定的。确定块地址的工作一般由链接器来完成。由于大部分系统一般包含多个不同类型的存储器（EPROM，RAM 等），采用块可以使用户更有效地利用目标存储器。所有的块可以独立地进行重定位，因此可以将不同的块分配至各种目标存储器中。例如，可以定义一个包含初始化程序的块，然后可以将它分配至包含 EPROM 的存储器映像中。图 6-2 也示出了目标文件中的块与目标系统中存储器的关系。

在代码生成工具程序中，除了汇编器和链接器等程序外，TI 还提供了 C 编译器。C 编译器对 C 程序编译后也产生已初始化块和未初始化块两种，具体的块名稍有不同，除了不使用.data 块之外，还产生一些新的块。例如 TMS320C54x 定点 C 编译器产生的已初始化块包括：.text 块、.cinit 块、.const 块和.switch 块，未初始化块包括：.bss 块、.stack 块和 .sysmem 块。各块的内容如表 6-3 所示。

表 6-3　C 编译器产生的常用块及其内容

名　称	内　容
.cinit	已初始化的全局和静态变量表
.const	字符串，已初始化的全局和静态常数变量
.switch	switch 语句的跳转表
.text	可执行代码和浮点常数
.bss	全局和静态变量
.stack	软件堆栈
.sysmem	malloc 函数的动态存储区



6.4.2　汇编器对块的处理

汇编器对块的处理功能主要是确定汇编语言程序的各部分属于哪个特定的块。汇编器借助六个伪指令来完成这种功能，如：.bss，.usect，.text，.data，.sect 及.asect，其中.bss 和.usect 表明创建未初始化块，而.text，.data，.sect 及.asect 则表明建立已初始化块。当然，如果在程序中未用任何伪指令来指示，则汇编器将把所有的程序块或数据块统一汇编至 .text 块中。

1．未初始化块

未初始化块主要用来在存储器中保留空间，它们通常分配到 RAM 中。这些块在目标文件中没有实际内容，只是保留空间而已。程序可以在运行时利用这些空间来建立和存储变量。

未初始化数据块是通过使用.bss 和.usect 伪指令来建立的。.bss 的作用是在.bss 块中保留空间，而.usect 则是在自定义的块中保留空间，两条伪指令的写法分别为：

.bss 符号,字数

符号 .usect “块名”，字数

其中，符号指向由 .bss 或 .usect 保留的第一个字，它对应于为变量保留空间的变量名，可以在其他任何块中被访问，也可用 .global 伪指令定义为全局符号。字数表示空间的大小，而块名则是程序员自己定义的名字。汇编器遇到 .text，.data，.sect 和.asect 伪指令时立即停止汇编至当前块中且开始汇编至相应的块中。汇编器遇到 .bss 和 .usect 伪指令并不结束当前块开始一个新块，它们只是暂时离开当前块。.bss 和.usect 伪指令可以出现在一个已初始化块中的任何地方，而不影响已初始化块的内容。

2．已初始化块

已初始化块包含可执行代码或已初始化数据。这些块的内容存储在目标文件中，当程序装入时存放在 DSP 存储器中。每个已初始化块可以独立地进行重定位，且可访问在其他块中定义的符号。链接器可自动解决块与块之间符号访问的问题。有四个伪指令通知汇编器将代码或数据存放在一个块中，这四个伪指令的写法分别是：

.text

```
.data
.sect    “块名”
.asect   “块名”，地址
```

在这四个伪指令中，.text 和.data 建立的块名就是 .text 和 .data，而后两个建立的是自定义的块，其中 .asect 建立的块具有绝对的地址，一般不建议使用。

当汇编器遇到上述伪指令时，它立即停止汇编至当前块中，且开始将随后的代码或数据汇编至相应的块中。块是通过迭代过程建立的，例如，当汇编器首次遇到一个.data 伪指令，.data 块是空的，.data 后面的语句被汇编到 .data 块中，直到遇到.text、.sect 或 .asect 为止。如果汇编器在后面又遇到 .data 伪指令，则将这些 .data 后面的语句加到已存在于 .data 块中语句的后面，这样建立的唯一的 .data 块可以在存储器中分配一个连续的空间。

3．自定义块

自定义块就是程序员自己建立的块，它与默认的.text，.data 和.bss 块一样使用，但它与默认块分开汇编。例如，重复使用 .text 在目标文件中只建立一个.text 块，链接后这个.text 块也作为一个单位分配到存储器中。有时候程序员想把一部分程序放至不同于.text 的存储器中，则须使用自定义块，对数据也可同样处理。

有三条伪指令可用来建立自定义块：.usect，.sect 和.asect，其中.usect 建立的块的用法类似于.bss 块，它是在 RAM 中为变量保留空间；而.sect 和.asect 建立包含代码或数据的块，其用法类似于.text 块和.data 块。.sect 建立的块是可重定位的，而.asect 伪指令则建立具有绝对地址的块。这三条伪指令的写法如下：

```
符号     .usect   “块名”，字数
         .sect    “块名”
         .asect   “块名”，地址
```

6.4.3 链接器对块的处理

链接器对块的处理具有两个功能：首先，它将 COFF 目标文件中的块用来建立程序块或数据块，将输入块组合起来以建立可执行的 COFF 输出模块；其次，链接器为输出块选择存储器地址。

链接器提供两个命令来完成上述功能：MEMORY 和 SECTIONS。MEMORY 命令定义目标系统的存储器，程序员可定义每一块存储器，指定每一块的起始地址和长度。SECTIONS 告诉链接器如何组合输入块以及在存储器何处存放输出块。若不用这两个命令，则链接器采用默认的分配算法。若要用这两条命令，则一般需要建立一个链接命令文件，扩展名为.cmd，因此，一般地，程序员先根据系统要求编写一个链接命令文件，并把它添加到工程文件中。

链接命令文件的格式如下面的例子所示。

【例 6.1】 链接命令文件 myfunc.cmd

```
MEMORY
{
PAGE 0: PROG: origin = 0x1000, length = 0xEFFF
PAGE 1: DATA: origin = 0x0300, length = 0x0D00
}

SECTIONS
```

```
{
.text:      PAGE = 0
.data:      PAGE = 1
.cinit:     PAGE = 0
.bss:       PAGE = 1
}
```

这个文件和 TMS320C2xx 的默认链接命令相同。对不同的 DSP 芯片，分配算法有所不同。TMS320C2xx 链接器的默认分配算法为：

（1）程序存储器的起始地址为 0x1000，长度为 0xEFFF；

（2）数据存储器的起始地址为 0x300，长度为 0x0D00；

（3）将.text 分配到起始地址为 0x1000 的程序存储器中；

（4）将.data 分配到起始地址为 0x300 的数据存储器中；

（5）将.cinit 分配到紧接着.text 的程序存储器中；

（6）将自定义的已初始化块分配到紧接着.data 的存储器中，若自定义块不止 1 个，则按自定义块在输入文件的先后顺序存放；

（7）将.bss 分配到紧接着.data 的数据存储器中；

（8）将自定义的未初始化块分配到紧接着.bss 的存储器中，若自定义块不止 1 个，则按自定义块在输入文件的先后顺序存放。

如果在默认分配时有两个以上的目标文件，每个文件中都包含.text，.data，.bss 和自定义块，则链接器首先将各个文件中具有相同名字的块组合起来，形成相应的输出块，然后再按上述算法分配存储器。

一般情况下，默认分配不能满足用户的实际需求。不采用默认分配算法时，一般须建立一个链接命令文件，并在链接命令文件中用 MEMORY 和 SECTIONS 命令定义存储空间以及块的分配方法。

链接生成的.map 文件中包含了各程序块和变量的具体地址，链接生成的 .out 文件可用 CCS 中的菜单命令 File->Load Program 装入到软件模拟器 Simulator 或者硬件仿真器 Emulator 中执行。



6.4.4　程序重定位

汇编器对每块汇编时都假定其起始地址为 0，每块中所有的重定位符号（标号）都是相对于 0 地址而言的。当然，实际上并不是所有块在存储器中都是以 0 地址起始的，因此链接器必须通过下列方法对每块进行重定位：

（1）分配块至存储器，使各块有合适的起始地址；

（2）调整符号值，使之对应于新的块地址；

（3）调整重定位后符号的参考值，以反映调整后的符号值。

链接器利用重定位项来调整符号的参考值。汇编器每次在可重定位符号被参考时建立一个重定位项，链接器则在符号被重定位之后利用这些项来修正参考值。例 6.2 包含了产生重定位项的一个代码段。

【例 6.2】产生重定位项的代码（TMS320C2xx）

```
1              .ref     X
2 0000         .text
3 0000 FF80    B        X          ;产生一个重定位项
  0001 0000!
4 0002 D001    LALK     Y          ;产生一个重定位项
  0003 0004'
5 0004 CE1F    Y:   IDLE
```

在这个例子中，符号 X 和 Y 都是可重定位的，X 在其他模块中定义，Y 在本模块的.text 块中定义，汇编之后，X 的值为 0（汇编器假定所有的未定义的外部符号的值为 0），而 Y 的值为 4（相对于.text 块的 0 地址）。汇编器为 X 和 Y 各产生一个重定位项。X 的参考是外部参考（在列表文件中用!表示），Y 的参考是内部定义的可重定位符号（用'表示）。

链接之后，假定 X 重定位至地址 100h，且.text 块重定位到起始地址 200h，则 Y 重定位后的值为 204h。链接器利用这两个重定位项在目标代码中补上这两个参考：

```
FF80      B        X    变为   FF80
0000                            0100
D001      LALK     Y    变为   D001
0004                            0204
```

COFF 目标文件中的每一块都有一个重定位项目表。表中包含了块中每一个可重定位参考的一个重定位项。链接器通常在使用过重定位项之后就将它去除以防止对输出块再次重定位（如果目标文件被重新链接或被装入）。没有重定位项的文件就是所有地址都是绝对地址的绝对文件，如果程序员要保留重定位项，激活链接器的 －r 选项即可。

在有些情况下，一块程序装入在某块存储器中但需在另一块存储器中运行。例如，程序中的一段关键代码装在低速 ROM 中，但需在速度更高的 RAM 中运行。此时，必须进行运行时的重定位。实现运行时的重定位的方法比较简单，可在链接器命令文件的 SECTIONS 中将块分配两次：一次设定装入地址，另一次则设定运行地址。例如：

```
.text:    load＝ROM, run＝RAM0
```



6.4.5 COFF 文件中的符号

COFF 文件包含一个存储程序中符号信息的符号表，链接器在进行重定位时要用到这个符号表。调试工具也可利用这个符号表提供符号级调试。

在一个模块中定义在另一个模块中引用的符号称外部符号。一般可用 .def 、.ref、.global 等伪指令来标明某一符号为外部符号。在一个源模块中，外部符号可以用.def 、.ref、.global 来定义：

定义（.def）：	在当前模块中定义，在其他模块中引用
参考（.ref）：	在当前模块中引用，在其他模块中定义
全局（.global）：	包含上面两者
复制（.copy）：	从其他文件中读取源程序语句，完成后马上返回当前文件。读取的语句显示在列表文件中
包含（.include）：	从其他文件中读取源程序语句，完成后马上返回当前文件。读取的语句不显示在列表文件中

下面的代码可说明上述定义：

```
.global     X              ;X 在本模块中定义，可在其他模块中引用
.global     Y              ;Y 在其他模块中定义，在本模块中引用
X:  ADD 056h               ;定义 X
    MPY     Y              ;引用 Y
```

对 X 定义 .global 表明 X 是一个在本模块中定义的外部符号，可以在其他模块中引用。而对 Y 定义 .global 则表明 Y 在本模块中没有定义而在其他模块中定义。汇编器将 X 和 Y 都放在目标文件的符号表中。

汇编器汇编时每遇到一个外部符号都在符号表中产生一项关于该符号的记录，汇编器也产生指向每块开始的特殊符号，链接器则利用这些符号重定位块中其他符号。

值得指出的是，汇编器并不对其他类型的符号产生符号表项目，因为链接器并不需要它们，例如程序标号并不包含在符号表中（除非用 .global 说明）。有时符号表中包含程序中每个符号的项目是有用的，因此若要对程序进行符号级调试，须在汇编时激活 -s 选项。



6.4.6　COFF 格式编程示例

例 6.3 给出了一个采用 COFF 格式编写的 TMS320C2xx 的汇编语言。

【例 6.3】COFF 格式的汇编编程（TMS320C2xx）：

```
        ;文件名：demo.asm
            .data                           ;汇编至.data 块
coeff       .word     011h，022h，033h

            .bss   var1,1                   ;在 .bss 块为 var1 变量保留 1 个字的空间
            .bss   buffer,10                ;在 .bss 块为 buffer 变量保留 10 个字的空间

ptr         .word     0123h                 ;汇编至 .data 块

            .text                           ;汇编至 .text 块
add:        LAC C     #0Fh
aloop:      SBLK      1
            BLEZ      aloop
            SACL      var1,0

            .data                           ;汇编至 .data 块
ivals       .word     0aah,0bbh,0cch

var2        .usect    "newvars",1           ;自定义一个数据块
inbuf       .usect    "newvars",7

            .text                           ;汇编至 .text 块
            ADD        # 0FFh

            .sect      "mydata"             ;自定义一个数据块
val    .word0100h,0200h
```

例 6.3 的汇编程序经汇编器汇编后得到名为 demo.obj 的目标文件，在这个目标文件中共有

五个块：

.text	包含两段程序代码
.data	包含 7 个初始化的数据字
mydata	自定义块，包含 2 个初始化数据字
.bss	在存储器中保留 11 个字的空间
newvars	自定义块，在存储器中保留 8 个字的空间

demo.obj 再经链接器链接就得到可执行文件 demo.out。

6.5 存储区分配示例

6.5.1 TMS320C2xx 链接命令文件示例

例 6.4 给出了 TMS320LF2407A 的一个链接命令文件实例。其中，MEMORY 中定义了可用的存储器空间，PAGE 0 定义了中断矢量和程序存储空间，PAGE 1 定义了存储器映射寄存器、内部数据存储空间和外部数据存储空间。MEMORY 中的 origin 表示某一块存储空间的起始地址，length 则表示该块存储空间的大小。SECTIONS 中规定了程序中各个块的分配空间，在该例中，.text 块分配到 PAGE 0 的 PROG 中，而.bss 块则被分配到 PAGE 1 的 B0 空间中。

在该例子的开始部分指定了链接生成的文件名，需要链接的库文件以及链接的目标文件，需要说明的是，在 CCS 中，这部分内容可不在链接命令文件中指定，而直接在 CCS 的编译器和链接器中设置。

【例 6.4】TMS320LF2407A 链接命令文件

```
/*****************************************************/
/*            TMS320LF2407A 链接命令文件          demo.cmd*/
/*****************************************************/
－c                    /*ROM 初始化模式*/
－o    demo.out       /*产生 demo.out 可执行文件*/
－m    demo.map       /*产生 demo.map 映像文件*/
－l    rts2xx.lib     /*TMS320C2xx 库函数*/
main.obj   /*链接的第一个目标文件*/
sub.obj    /*链接的第二个目标文件*/
asm.obi    /*链接的第三个目标文件*/
MEMORY
{
PAGE 0 :
VECS: origin =     0h , length =    040h      /*  中断矢量  */
PROG: origin =    44h , length = 0FFBCh      /*  程序  */
PAGE 1 :
MMRS: origin =     0h , length =    060h      /*  存储器映射寄存器*/
B2: origin = 0060h , length =    020h        /* B2 块：DARAM*/
B0: origin = 0200h , length =   0100h        /*B0 块：  DARAM*/
B1: origin = 0300h , length =   0100h        /*B1 块：DARAM* /
```

```
SARAM:origin = 0800h , length =    0800h     /*内部 SARAM 块*/
DATA: origin = 8000h , length =    8000h     /*外部数据存储器*/
}

SECTIONS
{
      .text:{}      > PROG PAGE 0      /*.text 块分配到 PROG 中*/
      .cinit:{}     > PROG PAGE 0      /*.cinit 块分配到 PROG 中*/
      .data:{}      > DATA PAGE 1      /*.data 块分配到 DATA 中*/
      .stack:{}     > SARAM PAGE 1     /*.stack 块分配到 SARAM 中*/
      .bss:{}       > B0     PAGE 1    /*.bss 块分配到 B0 中*/
      .vectors:{} > VECS PAGE 0       /*自定义的.vectors 块分配到 VECS 中*/
      .newvars:{} > B1     PAGE 1     /*自定义的.newvars 块分配到 B1 中*/
}
```



6.5.2　TMS320VC5416 链接命令文件示例

例 6.5 是 TMS320VC5416 的一个链接命令文件实例。与上例相同，MEMORY 中定义了可用的存储器空间，而 SECTIONS 中规定了程序中各个块的分配空间。

【例 6.5】TMS320VC5416 链接命令文件

```
-o    spcodec.out
-m spcodec.map

-l rts.lib
-stack 0x0C00

MEMORY
      {
          PAGE 0:       /* Program space */
      VECS        : origin = 0080h, length = 0080h
      PROG        : origin = 3000h, length = 5000h

          PAGE 1:       /* Data space */
      FFT_RAM     : origin = 0x200, length = 0x200
      DATA0       : origin = 0060h, length = 0020h    /* Scratch Pad mem space */
      STACK       : origin = 0400h, length = 0C00h    /*  堆栈  */
      DATA1       : origin = 1000h, length = 2000h
      }

SECTIONS
      {
      .vectors    : {} > VECS          PAGE 0       /*中断矢量表  */
      .text       : {} > PROG          PAGE 0       /*  程序代码  */
      .fftram     : {} > FFT_RAM       PAGE 1       /*自定义的.fftram 分配到 FFT_RAM 中*/
      .stack: {} > STACK               PAGE 1       /*.stack 块分配到 SARAM 中*/
      .data  : {} > DATA1              PAGE 1       /*.data 块分配到 DATA1 中*/
```

```
.bss   : {} > DATA1              PAGE 1      /*.bss 块分配到 DATA1 中*/
}
```

6.6 小　　结

本章介绍了 TI 公司汇编器和链接器所采用的 COFF 目标文件格式，着重介绍了这种文件格式中的最小单位 Section，讨论了汇编器和链接器对块的处理方法，并介绍了基于 COFF 的汇编语言编程。COFF 目标文件格式非常有利于模块化编程，使管理代码段和系统存储器更为方便和灵活。基于这种格式编写 DSP 程序可使程序员摆脱对程序绝对地址的操作，也使程序具有更强的可读性和可移植性。

6.7 习题与思考题

（1）什么是 COFF 格式？它有什么特点？
（2）说明 .text 块、.data 块 和.bss 块分别包含什么内容？
（3）链接命令文件包括哪些内容？
（4）什么是程序的重定位？
（5）已知一数组名为 indata，长度为 30 个字，试用两种方法给该数组分配空间。

第 7 章　TMS320C5000 DSP 芯片的汇编语言

TMS320C5000 系列芯片内部采用了算术逻辑单元和乘累加单元等多种专用硬件结构来提高 DSP 性能。基于 DSP 的硬件结构，TI 提供了完善的指令集来提高芯片的处理能力。TMS320C5000 系列芯片可采用两种不同的指令系统：助记符指令集和代数指令集。本章以助记符指令集为重点，以 C54x 为例介绍 DSP 的汇编语言指令，与助记符指令相对应的代数指令集在附录 A 中列出。7.1 节首先介绍汇编语言源程序的格式，由于指令集涉及许多符号与缩写，因此 7.2 节对有关的符号和缩写进行说明，7.3 节对汇编指令系统作一总体介绍，详细说明汇编语法和指令使用方法，7.4 节重点介绍 TMS320C5000 的寻址方式。

7.1　汇编语言源程序格式

汇编语言是 DSP 应用软件的基础，编写汇编语言必须要符合相应的格式，这样汇编器才能将源文件转换为机器语言的目标文件。C54x 汇编语言源程序由源语句组成，包含汇编语言指令、汇编伪指令、宏伪指令和注释等，一般一句程序占据编辑器的一行。由于汇编器每行最多只能读 200 个字符，所以源语句的字符数不能超过 200。一旦长度超过 200 个字符，汇编器将自行截去行尾的多余字符并给出警告信息。如果截去的是注释，并不影响程序的正确执行；但如果截去了语句的执行部分，则程序会编译出错或错误执行。

汇编语句格式可以包含 4 个部分：标号域、指令域、操作数域和注释域，以助记符指令为例，格式如下：

[标号][：]　　指令[操作数列表]　　[；注释]

其中[]内的部分是可选项。在编写汇编指令时，必须遵循以下格式：

（1）语句必须以标号、空格、星号或分号开始；

（2）标号为可选项，若要使用标号，则必须从第 1 列开始。标号长度最多为 32 个字符，由 A~Z，a~z，0~9，_和$等组成，但第 1 个字符不能为数字。标号后可以跟一个冒号（：），但并不作为标号的一部分；

（3）每个域必须由 1 个或多个空格分开，制表符等效于空格；

（4）注释是可选项，开始于第 1 列的注释须用星号或分号（*或；）标示，但在其他列开始的注释前面只能标分号；

（5）指令域一定不能从第 1 列开始，否则将被视为标号。指令域包括以下操作码之一：助记符指令（详见 7.3 节）、汇编伪指令（如.data，.set）、宏伪指令（如.var，.macro）和宏调用；

（6）操作数域为操作数的列表。操作数可以为常数，符号或者是表达式。当操作数为立即寻址时，使用＃符号作为前缀；操作数为间接寻址时，使用*符号作为前缀，将操作数的内容作为地址。

7.2 汇编源程序中常见符号和伪指令

为了便于学习和应用，表 7-1 列出了指令集中所用到的符号与缩写。

表 7-1 指令集符号与缩写

符号	说 明	符号	说 明
A	累加器 A	n	XC 指令操作数。n=1 表示 1 条指令执行；n=2 表示 2 条指令执行
ALU	算术逻辑单元	N	指定 RSBX/SSBX/XC 指令中修改的状态寄存器。N=0，状态寄存器 ST0；N=1，状态寄存器 ST1
AR	辅助寄存器	OVA	ST0 中的累加器 A 的溢出标志
ARx	指定的辅助寄存器（0≤x≤7）	OVB	ST0 中的累加器 B 的溢出标志
ARP	ST0 中的辅助寄存器指针域，3 位域指针指向当前的辅助寄存器（AR）	OVdst	目的累加器（A 或 B）的溢出标志
ASM	ST1 中的 5 位累加器移位模式域（-16≤ASM≤15）	OVdst_	与当前目的累加器相反的累加器的溢出标志
B	累加器 B	OVsrc	源累加器（A 或 B）的溢出标志
BRAF	ST1 中的块重复有效标志	OVM	ST1 中的溢出模式位
BRC	块重复计数器	PA	16 位立即数表示的端口地址（0≤PA≤65535）
BITC	4 位值，用于确定位测试指令对指定数据存储区的哪一位进行测试	PAR	程序地址寄存器
C16	ST1 中的双 16 位/双精度算术模式比特	PC	程序计数器
C	ST0 中的进位位	pmad	16 位立即数表示的程序存储区地址（0≤pmad≤65535）
CC	2 位的条件代码	Pmem	程序存储区操作数
CMPT	ST1 中的兼容模式位	PMST	处理器模式状态寄存器
CPL	ST1 中的编译模式位	prog	程序存储区操作数
cond	条件执行指令所用的条件	[R]	舍入（凑整）选项
[d],[D]	延迟方式的选项	rnd	舍入
DAB	地址总线	RC	循环计数器
DAR	DAB 地址寄存器	RTN	在 RETF[D]指令中使用的快速返回寄存器
dmad	16 位立即数表示的数据存储区地址（0≤dmad≤65535）	REA	块循环尾地址寄存器
Dmem	数据存储区操作数	RSA	块循环起始地址寄存器
DP	ST0 中的 9 位数据存储区页指针域	SBIT	4 位域指示在 RSBX/SSBX/XC 指令中修改的状态寄存器位数（0≤SBIT≤15）
dst	目的累加器（累加器 A 或 B）	SHFT	4 位指示的移位数（0≤SHFT≤15）
dst_	与当前目的累加器相反的累加器。 若 dst=A,dst_=B；若 dst=B,dst_=A	SHIFT	5 位指示的移位数（-16≤SHIFT≤15）
EAB	E 地址总线	Sind	使用间接寻址的单访问数据存储区操作数
EAR	EAB 地址寄存器	Smem	16 位单访问的数据存储区操作数
extpmad	23 位立即数表示的程序存储区地址	SP	堆栈指针
FRCT	ST1 中的分数模式位	src	源累加器（A 或 B）
hi(A)	累加器 A 中的高 16 位（31~16 位）	ST0,ST1	状态寄存器 0，状态寄存器 1
HM	ST1 中的保持模式位	SXM	ST1 中的符号扩展模式位
IFR	中断标志寄存器	T	暂存寄存器
INTM	ST1 中的中断屏蔽位	TC	ST0 中的测试/控制标志位
K	小于 9 位的短立即数	TOS	堆栈栈顶
k3	3 位立即数（0≤k3≤7）	TRN	转换寄存器
k5	5 位立即数（0≤k5≤7）	TS	T 寄存器中位（5~0）确定的移位数

（续表）

符号	说　明	符号	说　明
k9	9 位立即数（0≤k9≤7）	uns	无符号数
lk	16 位长立即数	XF	ST1 中的外部标志状态位
Lmem	使用长字寻址的 32 位单访问数据存储区操作数	XPC	程序计数器的扩展寄存器
mmr MMR	存储区映射寄存器	Xmem	在双操作数和一些单操作数指令中使用的 16 位双访问数据存储区操作数
MMRx MMRy	存储区映射寄存器，AR0~AR7 或 SP	Ymem	在双操作数指令中使用的 16 位双访问数据存储区操作数
（　）	寄存器或存储单元包含的内容	#	立即数寻址时常数的前缀
[　]	表示可选项	‖	并行指令

汇编程序支持一些 C 编译器用于符号调试的伪指令：

- .func 和.endfunc 指明 C 函数的开始和终止。
- .block 和.endblock 指明 C 块的边界。
- .file 用来定义符号表中的一个符号，确定当前的源文件名。
- .line 用于指明当前属于 C 源程序中哪一行。
- .sym 指明全局变量、局部变量或函数的符号调试信息。其格式如下：

.sym　name, value [, type, storage class, size, tag, dims]

其中，name 放入符号表的变量名，前 32 字符有效；value 指定变量的值，可用合法的表达式表示；type 指定变量的 C 语言类型；storage class 指定变量的 C 存储类型，如结构、枚举、联合、外部定义、外部变量、静态变量等；size 指明该变量的比特位数；tag 指明这个变量所属类型或结构的名称，这个名称必须已经由.stag、.etag 或.utag 声明过；dims 由 1 到 4 个由逗号分隔的表达式组成，分别表明该变量的维数。

例如：

```
struct s { int member1, member2; } str;
int ext;
int array[5][10];
long *ptr;
int strcmp（）;

main（arg1,arg2）
int arg1;
char *arg2;
{
      register r1;
}

.global _array
.bss   _array,50,0,0
.sym  _array,_array,244,2,800,,5,10
.global      _ptr
.bss   _ptr,1,0,0
.sym  _ptr,_ptr,21,2,16
```

```
.global     _str
.bss   _str,2,0,0
.sym  _str,_str,8,2,32,_s
.global     _ext
.bss   _ext,1,0,0
.sym  _ext,_ext,4,2,16
```

- .stag、.etag 和.utag 分别定义结构、枚举和联合；.member 用于详指其中的一个成员；.eos 与.stag、.etag 和.utag 配合，表示该数据结构类型的终止点。其格式如下：

```
.stag name [, size]
.member
      .member
     ……
.eos

.etag name [, size]
.member
      .member
     ……
.eos

.utag name [, size]
.member
      .member
     ……
.eos

.member name, value [, type, storage class, size, tag, dims]
```

其中的 name 仍表示前 32 个字符有效。size 指定占用的存储空间的比特位，可以省略。

一般编译器并不在汇编程序中产生并列出这些符号调试伪指令。如果想要被列出，同时还保存汇编语言文件，可以在编译语句中加入-g 和-k 选项。

除去以上的伪指令外，汇编程序中还会涉及其他一些功能的伪指令。表 7-2 中列出了不同伪指令的助记符、语法格式及功能描述。

表 7-2　常见的伪指令

	助记符及语法	描　述
初始化常数	.byte　$value_1$[,…,$value_n$] .char　$value_1$[,…,$value_n$]	在当前块中以连续的方式放置一个或多个 8-bit 的数值，每个数值放置入一个字中，高 8 比特用 0 填充
	.short　$value_1$[,…,$value_n$] .int　$value_1$[,…,$value_n$] .word　$value_1$[,…,$value_n$]	在当前块中以连续的方式放置一个或多个 16-bit 的数值，每个数值放置入一个字中
	.long　$value_1$[,…,$value_n$]	在当前块的两个连续字中放置 32-bit 的数值；重要字先存储
	.string　$value_1$[,…,$value_n$]	在当前块中放置 8-bit 的字符；一个字放置一个 8-bit 字符
	.float　$value_1$[,…,$value_n$]	将单精度 IEEE 浮点数用连续的双字方式放置入当前块；重要字先存储
	.field　value[, size in bits]	将一个单值放入一个字的指定位中；可用于在一个字中配置多个域；直到一个字配置满，编译器才指向下一个字配置；若前面加标识，则该标识指向放置该域的字

（续表）

	助记符及语法	描　述
	.space　size in bits	在当前块中保留指定数量的位；汇编器在这些保留的位填 0；也可以通过用 16 乘以需保留的字数来保留字；当前面有标号时，其指向保留位的第一个字或字节
地址对齐	.align [size]	使 SPC（section program counter）指向 1 到 128 个字的边界；Size 的值必须为 2 的整数幂，默认时相当于 128 .align　1　　；SPC 指向字的边界； .align　2　　；SPC 指向双字（偶数地址）的边界； .align　[128]　　；SPC 指向页的边界
	.even	使 SPC 指向下一个偶数地址字的边界；等价于.align　2
符号定义	.asg　[“]character　string[”], substitution symbol	将一个字符串赋给一个替代符
	.equ / .set	使符号等于一个数；两者一样，可以交换使用
	.union	开始共同体定义
	.endunion	结束共同体定义
	.struct	开始结构体定义
	.endstruct	结束结构体定义
条件控制	.if　well-defined expression	如果条件成立，汇编该块代码
	.else	如果.if 条件不成立，汇编该块代码
	.elseif well-defined expression	如果.if 条件不成立，.elseif 条件成立，汇编该块代码
	.endif	结束.if 代码块
	.loop [well-defined expression]	开始代码块的重复汇编，后跟循环次数
	.break[well-defined expression]	如果条件成立，结束.loop 汇编
	.endloop	结束.loop 代码块
宏定义	.macro	定义宏；可在程序任意部分定义，但在使用宏前必须已定义
	.mlib　[“]filename[”]	定义宏库
	.mexit	转到.endm
	.endm	终止宏定义
	.var	定义一个局部宏替代符

7.3　汇编指令系统

C54x 的指令集包含了 4 种基本类型的操作：

（1）算术指令。包括加法、减法、乘法、乘累加、乘累减、32 位操作数运算指令以及其他一些专用指令。

（2）逻辑指令。包括与、或、异或、移位和测试指令等。

（3）程序控制指令。包括跳转、调用、中断、返回、重复等程序控制指令。

（4）装载和存储指令。包括装载、存储以及并行装载和存储的指令等。

为了便于学习和查阅方便，下面将按字母顺序依次对汇编指令进行介绍，讲解指令的语法和执行过程，并举例说明。在附录 A 中，将按照指令的 4 种不同功能分类，对助记符指令和代数指令的语法进行对照总结。

■　ABDST

语法：ABDST　Xmem，Ymem

说明：循环使用该条指令可以计算两个矢量的距离（即分量差的绝对值和）。首先将累加器 A 的高端（32~16 位）的绝对值加到累加器 B 中。然后 Xmem 减去 Ymem 之差左移 16

位并存放在累加器 A 中。指令受 OVM，FRCT，SXM 状态标志位的影响，其中 FRCT=1 时绝对值需乘 2。执行结果影响 C，OVA 和 OVB。

举例：

ABDST　*AR3+，*AR4+

	Before Instruction	After Instruction
A	FF ABCD 0000	FF FFAB 0000
B	00 0000 0000	00 0000 5433
AR3	0100	0101
AR4	0200	0201
FRCT	0	0
Data Memory		
0100h	0055	0055
0200h	00AA	00AA

■ ABS

语法：ABS　src　[, dst]

说明：计算 src 的绝对值，结果放入 dst 中。如果没有定义 dst，绝对值放入 src 中。指令受 OVM 状态标志位的影响：若 OVM＝1，则 80 0000 0000h 的绝对值为 00 7F FFFFh；若 OVM＝0，则 80 0000 0000h 的绝对值为 80 0000 0000h。执行结果影响 C 和 OVdst（若未指定 dst，则为 OVsrc）。

举例：

ABS　A，B

	Before Instruction		After Instruction	
A	FF FFFF FFCB	−53	FF FFFF FFCB	−53
B	FF FFFF FC18	−1000	00 0000 0035	+53

■ ADD

语法：
```
1:    ADD    Smem, src
2:    ADD    Smem,  TS,  src
3:    ADD    Smem,  16,  src  [, dst]
4:    ADD    Smem  [,SHIFT],  src [, dst]
5:    ADD    Xmem,  SHFT,  src
6:    ADD    Xmem,  Ymem,  dst
7:    ADD    #lk  [, SHFT], src  [, dst]
8:    ADD    #lk,  16,  src  [, dst]
9:    ADD    src  [, SHIFT]  [, dst]
10:   ADD    src,  ASM  [, dst]
```

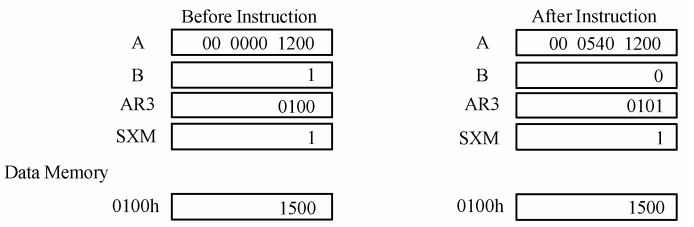
说明：将一个 16 位的数加到选定的累加器中。这个 16 位数可以为下列情况之一：

- 单访问的数据存储区操作数 Smem；
- 双访问的数据存储区操作数 Ymem；
- 立即数#lk；
- 累加器 src 移位后的值。

若定义了 dst，加法结果存入 dst，否则存入 src 中。操作数左移时低位加 0，右移时，若 SXM＝1，则高位进行符号扩展；若 SXM＝0，则高位加 0。指令受 OVM 和 SXM 状态标志位的影响，执行结果影响 C 和 OVdst（若未指定 dst，则为 OVsrc）。

举例：

ADD　*AR3+，14，A



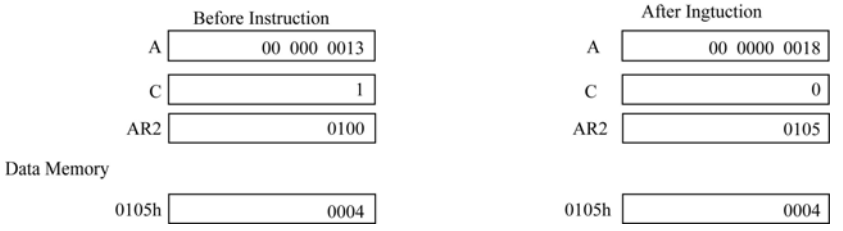
■　ADDC

语法：ADDC　Smem，　src

说明：将 16 位单访问操作数 Smem 和进位位 C 的值加到 src 中，结果存入 src。指令受 OVM 和 C 状态标志位的影响，执行结果影响 C，OVsrc。

举例：

ADDC　*AR2(5).A



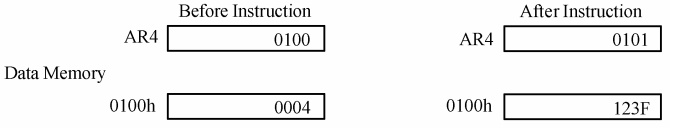
■　ADDM

语法：ADDM　#lk,　Smem

说明：Smem 与 16 位立即数相加，结果存放在 Smem 中。该指令不能循环执行。指令受 OVM 和 SXM 状态标志位的影响，执行结果影响 C 和 OVA。

举例：

ADDM 0123Bh,*AR4+



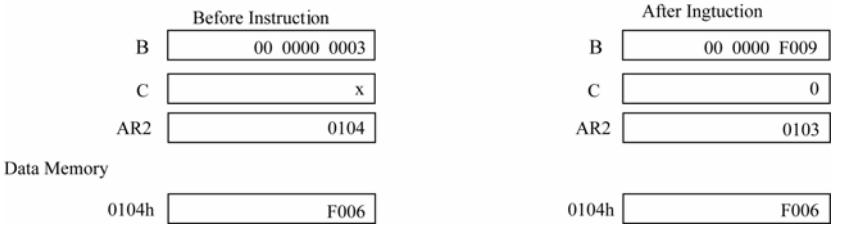
■　ADDS

语法：ADDS　　Smem，　src

说明：将无符号 Smem 与 src 相加，结果存入 src。无论 SXM 为何值均不进行符号扩展。指令受 OVM 状态标志位的影响，执行结果影响 C 和 OVsrc。

举例：

ADDS　*AR2-.B



■ AND

语法：1: AND Smem, src

2: AND #lk [, SHFT], src [, dst]

3: AND #lk, 16, src [, dst]

4: AND src [, SHIFT] [, dst]

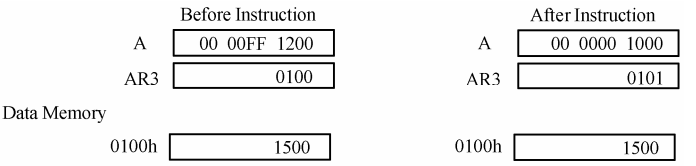
说明：将数据与 src 进行与运算。数据为以下几种情况之一：

- 16 位单访问 Smem；
- 16 位立即数 lk；
- 源累加器 src 或目的累加器 dst。

如果指令中有移位，则操作数移位后再进行与操作。左移时低位清零，高位无符号扩展；右移时高位也不进行符号扩展。

举例：

AND *AR3+,A



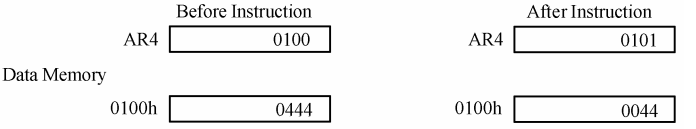
■ ANDM

语法：ANDM #lk, Smem

说明：Smem 与 16 位立即数 lk 进行与运算，结果放入原 Smem 所在单元。该指令不能循环执行。

举例：

ANDM #00FFh,*AR4+



■ B[D]

语法： B[D] pmad

说明：将 pmad 指定的程序存储区地址赋给 PC，实现分支跳转。pmad 可以为符号或者数字。若带后缀 D，为延迟方式，此时紧跟该指令的两条单字指令或 1 条双字指令先被取出执行，然后程序再跳转。该指令不能被循环执行。

举例：

B 2000h



■ BACC[D]

语法： BACC[D] src

说明：src 低 16 位所确定的地址赋给 PC。若为延迟方式，此时紧跟该指令的两条单字指令或 1 条双字指令先被取出执行，然后程序再跳转。该指令不能被循环执行。

举例：

```
BACCD B
ANDM 4444h,*AR1+
```



执行流程为：操作数与 4444h 与运算后，程序从 2000h 单元继续执行。

■　BANZ[D]

语法：BANZ[D]　pmad,　Sind

说明：若当前辅助寄存器 ARx 不为 0，则 pmad 值赋给 PC，否则 PC 值加 2。若为延迟方式，此时紧跟该指令的两条单字指令或 1 条双字指令先被取出执行，然后程序再跳转。该指令不能被循环执行。

举例：

```
BANZ 2000h,*AR3-
```



■　BC[D]

语法：　BC[D]　pmad,　cond[, cond　[, cond]]

说明：如果满足给定的条件，pmad 值赋给 PC 实现条件转移。若为延迟方式，此时紧跟该指令的两条单字指令或 1 条双字指令先被取出执行，且不会影响被测试的条件。若条件不满足，PC 加 2，条件满足时程序跳到 pmad 地址处。指令不能循环执行。表 7-3 列出了指令可能对应的各种条件。

表 7-3　指令对应的各种条件

条 件	说　明	条 件	说　明
BIO	$\overline{\text{BIO}}$ 引脚为低电平	NBIO	$\overline{\text{BIO}}$ 引脚为高电平
C	C=1	NC	C=0
TC	TC=1	NTC	TC=0
AEQ	（A）=0	BEQ	（B）=0
ANEQ	（A）≠0	BNEQ	（B）≠0
AGT	（A）>0	BGT	（B）>0
AGEQ	（A）≥0	BGEQ	（B）≥0
ALT	（A）<0	BLT	（B）<0
ALEQ	（A）≤0	BLEQ	（B）≤0
AOV	A 溢出	BOV	B 溢出
ANOV	A 没有溢出	BNOV	B 没有溢出
UNC	无条件的		

指令可以同时对 1 个、2 个或 3 个条件进行测试。在对超过 1 个的多个条件测试时，这些条件只能出自表 7-4 所示的同一组的不同类中。

例如：因为 TC、C 和 BIO 同组不同类，因此可以同时测试 TC、C 和 BIO，但不能同时测试 NTC、C 和 NC。

当测试条件为 OV 或 NOV 时，结果影响 OVA 或 OVB。

表 7-4　测试条件的分组分类

组 1		组 2		
A 类	B 类	A 类	B 类	C 类
EQ NEQ LT LEQ GT GEQ	OV NOV	TC NTC	C NC	BIO NBIO

举例：

```
BC 2000h,AGT
```

Before Instruction		After Instruction	
A	00 0000 0053	A	00 0000 0053
PC	1000	PC	2000

■ BIT

语法：BIT　Xmem,　BITC

说明：将 Xmem 的指定位复制到状态寄存器 ST0 的 TC 位。表 7-5 列出了对应于 Xmem 每一位的位代码。从表中可以看出，位代码对应于 BITC，而位地址对应于（15-BITC）。结果影响状态标志位 TC。

表 7-5　BIT 指令的位代码

位地址	位代码	位地址	位代码
（低位）0	1111	8	0111
1	1110	9	0110
2	1101	10	0101
3	1100	11	0100
4	1011	12	0011
5	1010	13	0010
6	1001	14	0001
7	1000	15	0000

举例：

```
BIT *AR5+,15-12;test bit 12
```

Before Instruction		After Instruction	
AR5	0100	AR5	0101
TC	0	TC	1
Data Memory			
0100h	7688	0100h	7688

■ BITF

语法：BITF　Smem,　#lk

说明：测试 Smem 中由常数 lk 指定的某些位。如果这些指定位为 0，则状态寄存器 ST0 的 TC 位清零；否则 TC=1。

举例：

```
BITF 5,00FFh
```

Before Instruction		After Instruction	
TC	x	TC	0
DP	004	DP	004
Data Memory			
0205h	5400	0205h	5400

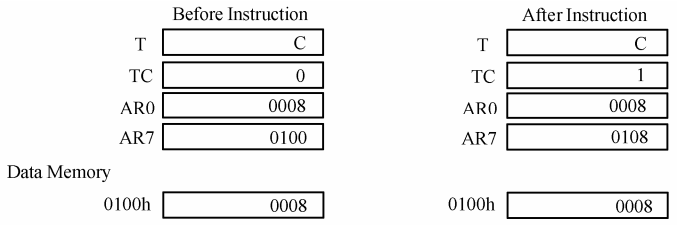
■　BITT

语法：BITT　Smem

说明：将 Smem 的指定位复制到状态寄存器 ST0 的 TC 位。被复制的位代码（见表 7-5）存放于 T 寄存器的低 4 位中，位地址对应于（15－T（3~0））。

举例：

BITT *AR7+0



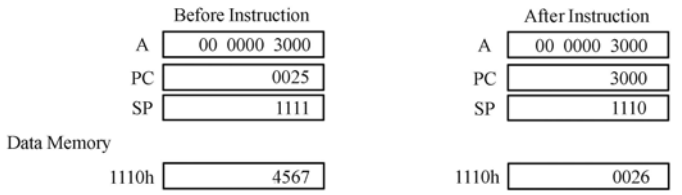
■　CALA[D]

语法：　CALA[D]　　src

说明：首先将返回地址压入栈顶，然后将 src 的低 16 位赋给 PC 实现调用。如果是延迟方式，紧接着该指令的两条单字指令或 1 条双字指令先被取出执行。该指令不能循环执行。

举例：

CALA　A



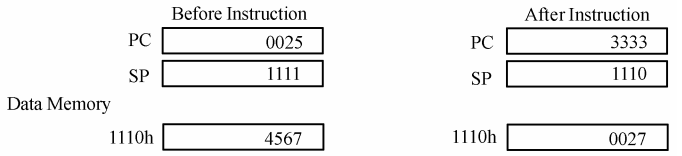
■　CALL[D]

语法：CALL[D]　pmad

说明：首先将返回地址压入栈顶保存，然后将 pmad 值赋给 PC 实现调用。如果是延迟方式，紧接着该指令的两条单字指令或 1 条双字指令先被取出执行。该指令不能循环执行。

举例：

CALL 3333h



■　CC[D]

语法：CC[D]　pmad,　cond　[, cond　[, cond]]

说明：如果满足指定的条件，则将返回地址压入栈顶，把 pmad 值赋给 PC 实现调用。若不满足条件，则 PC 加 2。如果为延迟方式，则紧接着该指令的两条单字指令或 1 条双字指令先被取出执行，且不会影响被测试的条件。该指令可以测试多个条件，条件代码及所对应的条件见表 7-3，条件的分组分类情况见表 7-4。当测试条件为 OV 或 NOV 时，指令执行结果影响 OVA 或 OVB。

举例：

CC 2222h,AGT

	Before Instruction	After Instruction
A	00 0000 3000	00 0000 3000
PC	0025	2222
SP	1111	1110
Data Memory		
1110h	4567	0027

■ CMPL

语法：CMPL src [, dst]

说明：计算 src 的反码（逻辑反），并将结果存放于 dst 中。如果没有指定 dst 则结果存放于 src 中。

举例：

CMPL A,B

	Before Instruction	After Instruction
A	FC DFFA AEAA	FC DFFA AEAA
B	00 0000 7899	03 2005 5155

■ CMPM

语法：CMPM Smem, #lk

说明：比较 Smem 与常数 lk 是否相等。若相等，TC=1，否则 TC=0。

举例：

CMPM *AR4+,0404h

	Before Instruction	After Instruction
TC	1	0
AR4	0100	0101
Data Memory		
0100h	4444	4444

■ CMPR

语法：CMPR CC, ARx

说明：根据条件代码 CC 比较指定的辅助寄存器 ARx 和 AR0 的值，根据比较结果决定 TC 值。条件代码 CC 的取值见表 7-6。如果满足条件，TC=1，否则 TC=0。注意所有的条件都以无符号操作数形式参与运算。

表 7-6 CMPR 指令的测试条件与代码

条件	条件代码（CC）	说明
EQ	00	测试 ARx 是否等于 AR0
LT	01	测试 ARx 是否小于 AR0
GT	10	测试 ARx 是否大于 AR0
NEQ	11	测试 ARx 是否不等于 AR0

举例：

CMPR 2,AR4

	Before Instruction	After Instruction
TC	1	0
AR0	FFFF	FFFF
AR4	7FFF	7FFF

■　CMPS

语法：CMPS　src,　Smem

说明：比较 src 高端（31~16 位）和低端（15~0 位）两个 16 位二进制补码的大小，将较大值放入 Smem 中。若高端数大，则 TRN 寄存器左移 1 位，最低位填 0，TC=0；若低端数大，则 TRN 寄存器左移 1 位，最低位填 1，TC=1。

举例：

CMPS A,*AR4+

	Before Instruction		After Instruction
A	00 2345 7899	A	00 2345 7899
TC	0	TC	1
AR4	0100	AR4	0101
TRN	4444	TRN	8889
Data Memory			
0100h	0000	0100h	7899

■　DADD

语法：DADD　Lmem,　src　[, dst]

说明：将 32 位操作数 Lmem 与 src 内容相加，加法结果放入 dst 中。如果没有指定 dst，则结果放入 src 中。32 操作数由两个连续地址的 16 位字构成，低地址必须为偶数，内容为 32 位操作数的高 16 位，高地址的内容则是 32 位操作数的低 16 位。注意 ST1 中的 C16 位决定了指令的方式：

若 C16=0，指令以双精度方式执行。40 位的 src 与 Lmem 相加，饱和与溢出位根据计算结果设置。

若 C16=1，指令以双 16 位方式执行。src 高端与 Lmem 的高 16 位相加；src 低端与 Lmem 的低 16 位相加。此时，饱和与溢出位不受影响。不管 OVM 位的状态如何，结果都不进行饱和运算。

当 C16＝0 时，指令受 OVM 和 SXM 状态标志位的影响，执行结果影响 C 和 OVdst（若未指定 dst，则为 OVsrc）。

举例：

DADD　*AR3+,A,B

	Before Instruction		After Instruction
A	00 5678 8933	A	00 5678 8933
B	00 0000 0000	B	00 6BAC BD89
C16	0	C16	0
AR3	0100	AR3	0102
Data Memory			
0100h	1534	0100h	1534
0101h	3456	0101h	3456

注：由于指令为长操作数指令，因此 AR3 执行后加 2。

■　DADST

语法：DADST　Lmem,　dst

说明：T 寄存器的值与 32 位 Lmem 相加，结果放入 dst 中。C16 的值决定了指令的运算方式：

若 C16=0，指令以双精度方式执行。此时 T 寄存器与其左移 16 位组成的 32 位数同 Lmem 相加，结果存放在 dst 中。

若 C16=1，指令以双 16 位方式执行。Lmem 的高 16 位与 T 寄存器值相加，结果存放在 dst 的高 24 位；同时，Lmem 的低 16 位与 T 寄存器相减，结果存放在 dst 的低 16 位。不管

OVM 状态如何，结果不进行饱和运算。

当 C16＝0 时，指令受 OVM 和 SXM 状态标志位的影响，执行结果影响 C 和 OVdst。

举例 1：

DADST *AR3-,A

	Before Instruction	After Instruction
A	00 0000 0000	00 3879 1111
T	2345	2345
C16	1	1
AR3	0100	00FE
Data Memory		
0100h	1534	1534
0101h	3456	3456

举例 2：

DADST *AR3+,A

	Before Instruction	After Instruction
A	00 0000 0000	00 3879 579B
T	2345	2345
C16	0	0
AR3	0100	0102
Data Memory		
0100h	1534	1534
0101h	3456	3456

■ DELAY

语法：DELAY　Smem

说明：将单访问数据存储区 Smem 的内容复制到下一个地址单元内，复制完后，原单元的内容保持不变。该指令常用于实现数字信号处理中的 Z 延迟。

举例：

DELAY *AR3

	Before Instruction	After Instruction
AR3	0100	0100
Data Memory		
0100h	6CAC	6CAC
0101h	0000	6CAC

■ DLD

语法：DLD　Lmem,　dst

说明：将 32 位 Lmem 放入 dst 中。若 C16=0，指令以双精度方式执行，Lmem 装入到 dst 中；若 C16=1，指令以双 16 位方式执行，Lmem 的高 16 位放入 dst 的高 24 位，Lmem 的低 16 位放入 dst 的低 16 位。指令受 SXM 状态标志位的影响。

举例：C16=0

DLD *AR3+,B

	Before Instruction	After Instruction
B	00 0000 0000	00 6CAC BD90
AR3	0100	0102
Data Memory		
0100h	6CAC	6CAC
0101h	BD90	BD90

■　DRSUB

语法：DRSUB　Lmem,　src

说明：32 位 Lmem 减去 src 内容，结果放入 src 中。指令执行方式与 C16 有关。

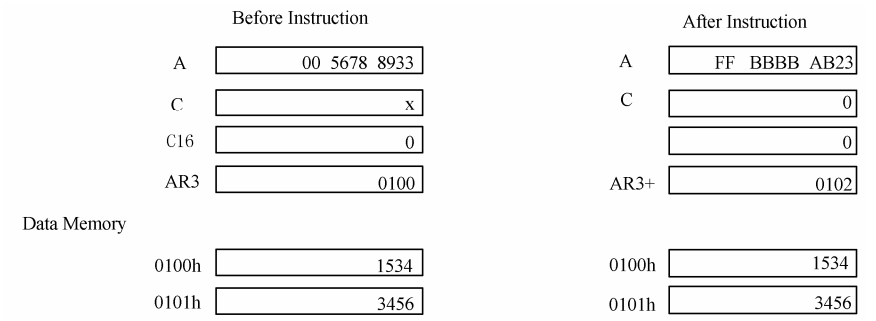
C16=0，指令以双精度方式执行。Lmem 的值减去 src（32 位），结果放入 src 中。

C16=1，指令以双 16 位方式执行。Lmem 的高 16 位减去 src 的高端（31~16 位），结果存放于 src 的前 24 位（39~16 位）；同时，Lmem 的低 16 位减去 src 的低端（15~0 位），结果存放于 src 的低端。不管 OVM 状态如何，结果都不进行饱和运算。

当 C16＝0 时，指令受 OVM 和 SXM 状态标志位的影响，执行结果影响 C 和 OVsrc。

举例：

```
DRSUB　*AR3+,A
```



■　DSADT

语法：DSADT　Lmem,　dst

说明：从 32 位 Lmem 中减/加 T 寄存器的内容，并将结果存放于 dst。C16 决定了指令的执行方式。

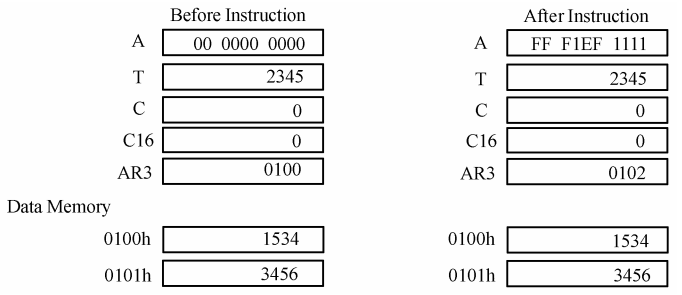
C16=0，指令以双精度方式执行，32 位的 Lmem 减去 T 寄存器与其左移 16 位组成的 32 位数，结果放入 dst。

C16=1，指令以双 16 位方式执行。Lmem 的高 16 位数减去 T 寄存器的内容，结果存放于 dst 的高端（39~16 位）；同时 Lmem 的低 16 位与 T 寄存器的内容相加，结果存放于 dst 的低端（15~0 位）。不管 OVM 状态如何，都不进行饱和运算。

当 C16＝0 时，指令受 OVM 和 SXM 状态标志位的影响，执行结果影响 C 和 OVdst。

举例 1：

```
DSADT *AR3+,A
```



举例 2：

```
DSADT *AR3-,A
```

	Before Instruction	After Instruction
A	00 0000 0000	FF F1EF 579B
T	2345	2345
C	0	1
C16	1	1
AR3	0100	00FE
Data Memory		
0100h	1534	1534
0101h	3456	3456

■ DST

语法：DST　src,　Lmem

说明：将 src 内容放入 32 位的 Lmem 中。

举例：

DST B,*AR3+

	Before Instruction	After Instruction
B	00 6CAC BD90	00 6CAC BD90
AR3	0100	0102
Data Memory		
0100h	0000	6CAC
0101h	0000	BD90

■ DSUB

语法：DSUB　Lmem,　src

说明：src 减去 32 位 Lmem，结果放入 src 中。C16 决定了指令执行的方式。

若 C16=0，指令以双精度方式执行。从 src 中减去 Lmem 的值。

若 C16=1，指令以双 16 位方式执行。从 src 的高端（31~16 位）减去 Lmem 的高 16 位，结果放入 src 的高 24 位（39~16 位）；同时 src 的低端（15~0 位）减去 Lmem 的低 16 位，结果放入 src 的低端。

当 C16＝0 时，指令受 OVM 和 SXM 状态标志位的影响，执行结果影响 C 和 OVsrc。

举例：

DSUB *AR3+,A

	Before Instruction	After Instruction
A	00 5678 8933	00 4144 54DD
C16	0	0
AR3	0100	0102
Data Memory		
0100h	1534	1534
0101h	3456	3456

■ DSUBT

语法：DSUBT　Lmem,　dst

说明：32 位 Lmem 减去 T 寄存器的值，结果放入 dst 中。其中 C16 决定了指令的执行方式。

C16=0，指令以双精度方式执行，32 位的 Lmem 减去 T 寄存器与其左移 16 位组成的 32 位数，结果放入 dst。

C16=1，指令以双 16 位方式执行。Lmem 的高 16 位数减去 T 寄存器的内容，结果存放于 dst 的高端（39~16 位）；同时 Lmem 的低 16 位减去 T 寄存器的内容，结果存放于 dst 的低

端（15~0 位）。不管 OVM 状态如何，都不进行饱和运算。

当 C16＝0 时，指令受 OVM 和 SXM 状态标志位的影响，执行结果影响 C 和 OVdst。

举例：

DSUBT *AR3-,A

	Before Instruction	After Instruction
A	00 0000 0000	FF F1EF 1111
T	2345	2345
C16	1	1
AR3	0100	00FE
Data Memory		
0100h	1534	1534
0101h	3456	3456

■　EXP

语法：EXP　src

说明：计算 src 的指数值并以二进制补码形式存放于 T 寄存器中。指数值通过计算 src 的引导位数并减 8 得到，范围在−8 到 31 之间。引导位数等于带 1 位符号位的数据左移至 40 位 src 的最高位时需要左移的位数。该指令不修改 src 的内容。

举例：

EXP A

	Before Instruction		After Instruction	
A	FF FFFF FFCB	−53	FF FFFF FFCB	−53
T	0000		0019	25

■　FB[D]

语法：FB[D]　extpmad

说明：将由 extpmad 的高 7 位（22~16 位）确定的页赋给 XPC，extpmad 的低 16 位赋给 PC，从而实现长跳转。若为延迟方式，此时紧跟该指令的两条单字指令或 1 条双字指令先被取出执行。该指令不能被循环执行。

举例：

FB 012000h

	Before Instruction	After Instruction
PC	1000	2000
XPC	00	01

■　FBACC[D]

语法：　FBACC[D]　src

说明：将 src 的 22~16 位赋给 XPC，并将 src 的 15~0 位赋给 PC，从而实现长跳转。若为延迟方式，此时紧跟该指令的两条单字指令或 1 条双字指令先被取出执行。该指令不能被循环执行。

举例：

FBACC A

	Before Instruction	After Instruction
A	00 0001 3000	00 0001 3000
PC	1000	3000
XPC	00	01

■　FCALA[D]

语法：FCALA[D]　src

说明：首先将返回地址 PC、XPC 压入堆栈，然后将 src 的 15~0 位赋给 PC，并将 src 的 22~16 位赋给 XPC，从而实现长调用。若为延迟方式，此时紧跟该指令的两条单字指令或 1 条双字指令先被取出执行。该指令不能被循环执行。

举例：

FCALA A

	Before Instruction	After Instruction
A	00 007F 3000	00 007F 3000
PC	0025	3000
XPC	00	7F
SP	1111	110F
Data Memory		
1110h	4567	0026
110Fh	4567	0000

注：由于该指令为单字指令，因此返回地址为 PC+1。

■ FCALL[D]

语法：FCALL[D] extpmad

说明：首先将返回地址 PC、XPC 压入堆栈，然后将 extpmad 的 15~0 位赋给 PC，并将 extpmad 的 22~16 位赋给 XPC，从而实现长调用。若为延迟方式，此时紧跟该指令的两条单字指令或 1 条双字指令先被取出执行。该指令不能被循环执行。

举例：

FCALL 013333h

	Before Instruction	After Instruction
PC	0025	3333
XPC	00	01
SP	1111	110F
Data Memory		
1110h	4567	0027
110Fh	4567	0000

注：由于该指令为双字指令，因此返回地址为 PC+2。

■ FIRS

语法：FIRS Xmem, Ymem, pmad

说明：该指令实现一个对称的有限冲激响应（FIR）滤波器。首先累加器 A 的高端（32~16 位）和由 pmad 寻址得到的 Pmem 相乘，乘法结果与累加器 B 相加并存放在累加器 B 中。同时，Xmem 和 Ymem 相加后的结果左移 16 位放入累加器 A 中。在下一个循环，pmad 加 1。一旦循环流水线启动，指令成为单周期指令。

指令受 OVM，FRCT 和 SXM 状态标志位的影响，执行结果影响 C，OVA 和 OVB。

举例：

FIRS *AR3+,*AR4+,COEFFS

	Before Instruction	After Instruction
A	00 0077 0000	00 00FF 0000
B	00 0000 0000	00 0008 762C
FRCT	0	0
AR3	0100	0101
AR4	0200	0201
Data Memory		
0100h	0055	0055
0200h	00AA	00AA
Program Memory		
COEFFS	1234	1234

■　FRAME

语法：FRAME　K

说明：将短立即数偏移 K 加到 SP 中。

举例：

FRAME 10h

	Before Instruction		After Instruction
SP	10000	SP	1010

■　FRET[D]

语法：FRET[D]

说明：长返回指令。首先将栈顶单元的低 7 位赋给 XPC，再把下一个单元的 16 位值赋给 PC。堆栈指针在每一操作完成后自动加 1。若为延迟方式，此时紧跟该指令的两条单字指令或 1 条双字指令先被取出执行。该指令不能被循环执行。

举例：

FRET

	Before Instruction	After Instruction
PC	2112	1000
XPC	01	05
SP	0300	0302
Data Memory		
0300h	0005	0005
0301h	1000	1000

■　FRETE[D]

语法：　FRETE[D]

说明：长中断返回指令。首先将栈顶单元的低 7 位赋给 XPC，再把下一个单元的 16 位值赋给 PC。程序从中断前的位置继续执行。该指令自动将 ST1 的中断屏蔽位（INTM）清零，允许中断。若为延迟方式，此时紧跟该指令的两条单字指令或 1 条双字指令先被取出执行。该指令不能被循环执行。

举例：

FRETE

	Before Instruction	After Instruction
PC	2112	0110
XPC	05	6E
ST1	xCxx	x4xxx
SP	0300	0302
Data Memory		
0300h	006E	006E
0301h	0110	0110

■　IDLE

语法：　IDLE　K

说明：该指令强迫程序执行等待操作直到产生非屏蔽中断或者复位操作。PC 值加 1，芯片保持空闲状态（低功耗模式）直至中断产生。即使 INTM=1，只要出现非屏蔽中断，DSP 也退出空闲状态。不管 INTM 如何取值，中断都通过中断屏蔽寄存器（IMR）使能。此时若 INTM=1，程序继续执行紧接着 IDLE 的指令；若 INTM=0，程序跳转到相应的中断服务子程

序。该指令不能循环执行。K 的取值在 1 到 3 之间，K 的不同取值确定了能让系统从空闲状态中释放的中断类型。

K=1，诸如定时器和串口等外围设备在空闲状态时仍然有效。外设中断、复位以及外部中断使 DSP 从空闲状态中释放出来。

K=2，诸如定时器和串口等外设在空闲状态下无效，复位、外部中断使 DSP 从空闲状态释放出来。由于在通常的设备操作时，中断在空闲状态无法锁存，因此中断必须持续多几个周期的低脉冲才被响应。

K=3，诸如定时器、串口等外设在空闲状态下无效，锁相环 PLL 停止。复位和外部中断使 DSP 从空闲状态中释放出来。由于在通常的设备操作时，中断在空闲状态无法锁存，因此中断必须持续多几个周期的低脉冲才被响应。

举例 1：

IDLE　1　DSP 保持空闲状态直至复位或非屏蔽中断产生。

举例 2：

IDLE　2　DSP 保持空闲状态直至复位或外部非屏蔽中断产生。

举例 3：

IDLE　3　DSP 保持空闲状态直至复位或外部非屏蔽中断产生。

■　INTR

语法：INTR　K

说明：首先将 PC 值压入栈顶，然后将由 K 所确定的中断向量赋给 PC，执行该中断服务子程序。中断标志寄存器（IFR）对应位清零且 INTM=1。该指令允许用户使用应用软件来执行任何中断服务子程序。注意中断屏蔽寄存器（IMR）不会影响 INTR 指令，并且不管 INTM 取值如何，INTR 指令都能执行。该指令不能循环执行。

举例：

INTR 3

	Before Instruction	After Instruction
PC	0025	FF8C
INTM	0	1
IPTR	01FF	01FF
SP	1000	0FFF
Data Memory		
0FFFh	9653	0026

■　LD

语法：1．LD　Smem,　dst

2．LD　Smem,　TS,　dst

3．LD　Smem,　16,　dst

4．LD　Smem　[, SHIFT],　dst

5．LD　Xmem,　SHFT,　dst

6．LD　#K,　dst

7．LD　#lk　[, SHFT],　dst

8．LD　#lk,　16,　dst

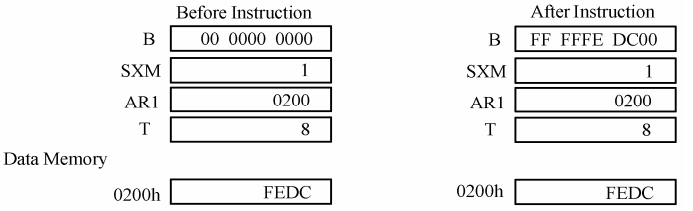
9．LD　src,　ASM　[, dst]

10．LD　src　[, SHIFT]　[, dst]

说明：将一个数据存储区操作数或一个立即数装入累加器 dst，若未确定 dst 时装入 src 中。该指令支持各种不同方式的移位操作。另外，指令还支持源累加器 src 内容移位后搬移到目的累加器 dst。注意所有的累加器装载指令都受 SXM 影响；有 SHIFT 或 ASM 移位的装载指令受 OVM 影响，同时影响 OVdst（dst=src 时影响 OVsrc）。

举例：

```
LD　*AR1,TS,B
```



■　LD

语法：1．LD　Smem,　T

2．LD　Smem,　DP

3．LD　#k9,　DP

4．LD　#k5,　ASM

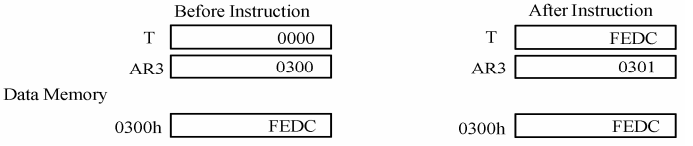
5．LD　#k3,　ARP

6．LD　Smem,　ASM

说明：将一个 Smem 或一个常数装入 T 寄存器，或者装入状态寄存器（ST0、ST1）中的 DP、ASM 和 ARP 域。

举例：

```
LD　*AR3+,T
```



■　LDM

语法：LDM　MMR,　dst

说明：将存储区映射寄存器 MMR 中的值装入 dst 的低 16 位，dst 的高 24 位（39~16 位）清 0。无论 DP 的当前值或 ARx 的高 9 位为何值，都将有效地址的高 9 位清零以限制在数据第 0 页寻址。该指令不受 SXM 的影响。

举例：

```
LDM　AR4,A
```



■　LD || MAC[R]

语法：LD　Xmem,　dst

|| MAC[R]　Ymem　[, dst_]

说明：并行指令。将 Xmem 左移 16 位放入 dst 高端（31~16 位）；同时 Ymem 与 T 寄存

器内容相乘，乘积与 dst_内容相加后存入 dst_。如果使用了 R 后缀，则还需进行舍入操作，此时乘累加结果与 2^{15} 相加，然后将低 16 位清零后的值存入 dst_中。指令受 OVM，FRCT 和 SXM 状态标志位的影响，执行结果影响 OVdst_。

举例：

```
LD   *AR4+,A
||MACR   *AR5+,B
```

	Before Instruction	After Instruction
A	00 0000 1000	00 1234 0000
B	00 0000 1111	00 010D 0000
T	0400	0400
FRCT	0	0
AR4	0100	0101
AR5	0200	0201
Data Memory		
0100h	1234	1234
0200h	4321	4321

■ LD || MAS[R]

语法：LD Xmem, dst

|| MAS[R] Ymem [, dst_]

说明：并行指令。将 Xmem 左移 16 位放入 dst 高端（31~16 位）；同时 Ymem 与 T 寄存器内容相乘，dst_减去该乘积，并将结果存入 dst_。如果使用了 R 后缀，则还需进行舍入操作，此时乘累减结果与 2^{15} 相加，然后将低 16 位清零后的值存入 dst_中。指令受 OVM，FRCT 和 SXM 状态标志位的影响，执行结果影响 OVdst_。

举例：

```
LD   *AR4+,A
||MAS   *AR5+,B
```

	Before Instruction	After Instruction
A	00 0000 1000	00 1234 0000
B	00 0000 1111	FF FEF3 8D11
T	0400	0400
FRCT	0	0
AR4	0100	0101
AR5	0200	0201
Data Memory		
0100h	1234	1234
0200h	4321	4321

■ LDR Smem, dst

语法：将 Smem 左移 16 位后装入 dst 高端（31~16 位），累加器 dst 的低 15 位（14-0 位）清零，第 15 位置为 1。指令受 SXM 的影响。

举例：

```
LDR *AR1,A
```

	Before Instruction	After Instruction
A	00 0000 0000	00 FEDC 8000
SXM	0	0
AR1	0200	0200
Data Memory		
0200h	FEDC	FEDC

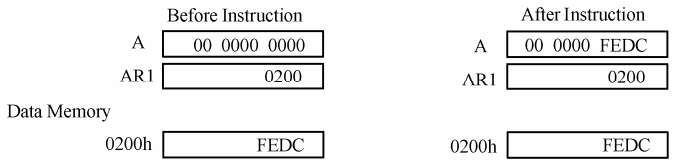
■　LDU

语法：LDU　Smem,　dst

说明：无符号数的装载指令。将 Smem 装入 dst 低端（15~0 位），dst 的保护位和高端（39~16 位）清零。无论 SXM 状态如何均无符号扩展。

举例：

LDU *AR1,A



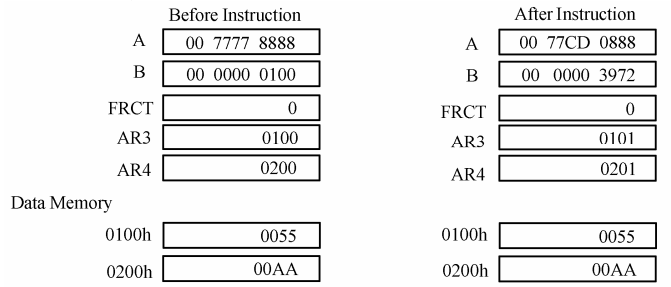
■　LMS

语法：LMS　Xmem,　Ymem

说明：该指令执行最小均方（LMS）算法。Xmem 左移 16 位与累加器 A 相加，其值再与 2^{15} 相加，结果放入累加器 A 中。同时并行执行 Xmem 与 Ymem 相乘，乘积与累加器 B 相加后，结果存放到累加器 B 中。由于 Xmem 不会覆盖 T 寄存器内容，因此 T 寄存器总存放着更新系数的误差值。指令受 OVM，FRCT 和 SXM 状态标志位的影响，执行结果影响 C，OVA 和 OVB。

举例：

LMS *AR3+,*AR4+



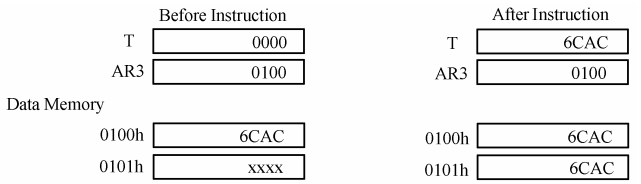
■　LTD　Smem

语法：LTD　Smem

说明：将 Smem 复制到 T 寄存器和紧接着的数据单元中。数据复制结束后，Smem 单元内容保持不变。该指令主要用于实现数字信号处理中的 Z 延迟。

举例：

LTD *AR3



■　MAC[R]

语法：1．MAC[R]　Smem,　src

2．MAC[R]　Xmem,　Ymem,　src　[, dst]

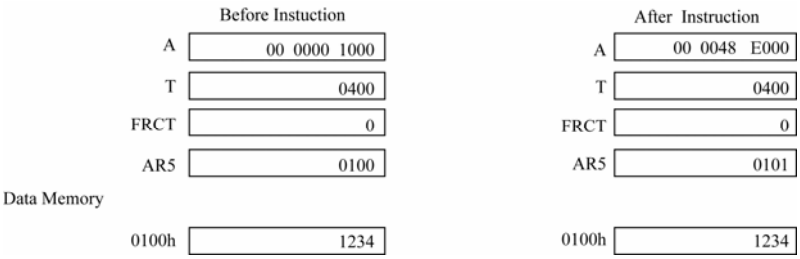
3．MAC　#lk,　src　[, dst]

4．MAC　Smem,　#lk,　src　[, dst]

说明：该指令实现乘累加运算，相乘结果和 src 相加，运算结果存放在 dst 中。若没有定义 dst，则存放在 src 中。对于第 1 种和第 3 种语法，其中一个乘数在 T 寄存器中。对于第 2 种语法，指令结束后，Xmem 中的值存放在 T 寄存器中。对于第 4 种语法，指令结束后，Smem 中的值存放在 T 寄存器中。如果使用了 R 后缀，则对乘累加值进行舍入处理。指令受 FRCT 和 SXM 状态标志位的影响，执行结果影响 OVdst。

举例：

MAC　*AR5+，A



■ MACA[R]

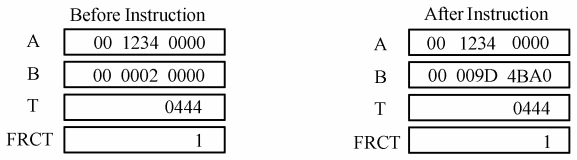
语法：1.　MACA[R]　Smem,　[, B]

2.　MACA[R]　T,　src　[, dst]

说明：Smem 或 T 寄存器中的内容与累加器 A 的高端（32~16 位）相乘，乘积与累加器 B（语法 1）或源累加器 src（语法 2）相加。乘累加结果存入累加器 B（语法 1），或者存入 dst（语法 2），若未定义 dst，则存入 src 中。如果使用了 R 后缀，则乘累加结果要进行舍入处理。对于语法 1，指令结束后，Smem 中的值存放在 T 寄存器中。指令受 OVM 和 FRCT 状态标志位的影响，语法 1 中执行结果影响 OVB，语法 2 中执行结果影响 OVdst（若未指定 dst，则为 OVsrc）。

举例：

MACA T,B,B



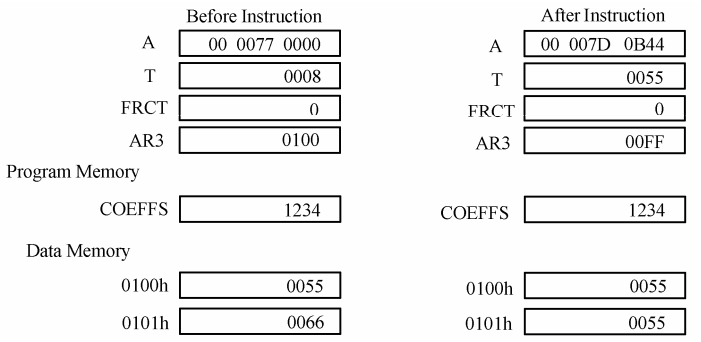
■ MACD

语法：MACD　Smem,　pmad,　src

说明：数据存储区 Smem 中的值与程序存储区 pmad 中的值相乘，乘积与 src 相加后存放在 src 中；同时将 Smem 值复制到 T 寄存器和紧接着 Smem 地址的数据单元中。由于指令将程序地址寄存器 PAR 中的程序存储区地址加 1，因此如果循环执行该指令，在循环流水线启动后该指令将变成单周期指令。指令受 OVM 和 FRCT 状态标志位的影响，执行结果影响 OVsrc。

举例：

MACD *AR3-,COEFFS,A



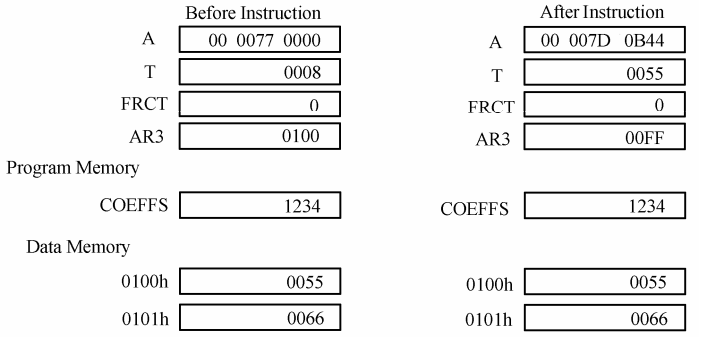
■　MACP

语法：MACP　Smem,　pmad,　src

说明：数据存储区 Smem 中的值与程序存储区 pmad 中的值相乘，乘积与 src 相加后存放在 src 中；同时将 Smem 值复制到 T 寄存器中。由于指令将程序地址寄存器 PAR 中的程序存储区地址加 1，因此如果循环执行该指令，在循环流水线启动后该指令将变成单周期指令。指令受 OVM 和 FRCT 状态标志位的影响，执行结果影响 OVsrc。

举例：

```
MACP *AR3-,COEFFS,A
```



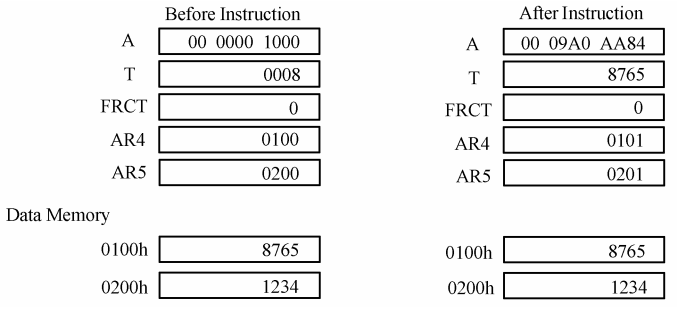
■　MACSU

语法：MACSU　Xmem,　Ymem,　src

说明：Xmem 中的 16 位无符号数与 Ymem 中 16 位有符号数相乘，乘积与 src 相加后放入 src。同时，Xmem 中的无符号数复制到 T 寄存器中。指令受 OVM 和 FRCT 状态标志位的影响，执行结果影响 OVsrc。

举例：

```
MACSU *AR4+,*AR5+,A
```



■ MAR

语法：MAR　Smem

说明：修改由 Smem 所确定的辅助寄存器的内容。该指令受兼容模式位 CMPT 的影响。

CMPT=1，若当前辅助寄存器 ARx 为 AR0，则指令修改 ARx（ARP）的内容，但不修改 ARP 的值；若当前辅助寄存器 ARx 不为 AR0，则修改 ARx 的内容，然后再将 x 值赋给 ARP。

CMPT=0，指令只修改 ARx 的内容，不修改 ARP。

举例 1：

MAR *AR3-

	Before Instruction		After Instruction
CMPT	1	CMPT	1
ARP	0	ARP	3
AR3	0100	AR3	00FF

举例 2：

MAR *AR0-

	Before Instruction		After Instruction
CMPT	1	CMPT	1
ARP	4	ARP	4
AR4	0100	AR4	00FF

■ MAS[R]

语法：1. MAS[R]　Smem,　src

2. MAS[R]　Xmem,　Ymem,　src　[, dst]

说明：Smem 与 T 寄存器内容相乘，或者 Xmem 与 Ymem 相乘，再从 src 中减去该乘积，结果存放在 dst 中。若未定义 dst，则存放在 src 中。同时，Xmem 值复制到 T 寄存器。如果使用了 R 后缀，则指令对结果进行舍入处理。指令受 OVM 和 FRCT 状态标志位的影响，执行结果影响 OVdst（若未指定 dst，则为 OVsrc）。

举例：

MASR *AR5+,*AR6+,A,B

	Before Instruction		After Instruction
A	00 0000 1000	A	00 0000 1000
B	00 0000 0004	B	FF F3B4 0000
T	0008	T	5678
FRCT	1	FRCT	1
AR5	0100	AR5	0101
AR6	0200	AR6	0201
Data Memory			
0100h	5678	0100h	5678
0200h	1234	0200h	1234

■ MASA[R]

语法：1. MASA　Smem　[, B]

2. MASA[R]　T,　src　[, dst]

说明：

语法 1：累加器 A 的高端（32~16 位）与 Smem 相乘，再从累加器 B 中减去该乘积，结果存放于累加器 B。同时，用 Smem 值更新 T 寄存器。

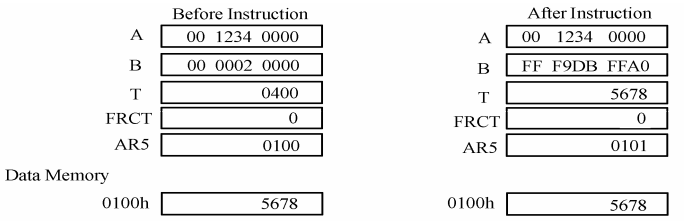
语法 2：累加器 A 的高端（32~16 位）与 T 寄存器的内容相乘，再从 src 中减去该乘积，

结果存放于 dst。若没有定义 dst，则存放于 src 中。如果使用了 R 后缀，则对结果进行舍入，将结果加 2^{15} 后低 16 位清零。

指令受 OVM 和 FRCT 状态标志位的影响，语法 1 中执行结果影响 OVB，语法 2 中执行结果影响 OVdst（若未指定 dst，则为 OVsrc）。

举例：

MASA *AR5+



■ MAX

语法：MAX　dst

说明：比较累加器 A、B 的内容，将最大值放入 dst 中。如果最大值在累加器 A 中，则进位位 C 清零，否则置 1。

举例：

MAX A



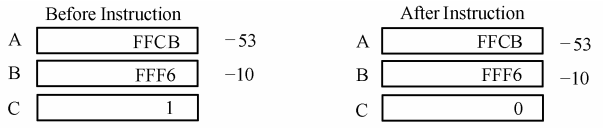
■ MIN

语法：MIN　dst

说明：比较累加器 A、B 的内容，将最小值放入 dst 中。如果最小值在累加器 A 中，则进位位 C 清零，否则置 1。

举例：

MIN A



■ MPY[R]

语法：1. MPY[R]　Smem,　dst

2．MPY　Xmem,　Ymem,　dst

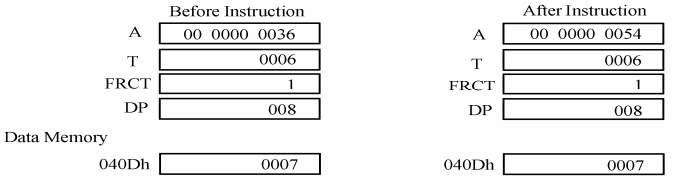
3．MPY　Smem,　#lk,　dst

4．MPY　#lk,　dst

说明：T 寄存器值或一个数据存储区值与另一个数据存储区值或立即数相乘，结果存放于 dst 中。在语法 2、3 中，在读操作数阶段要将 Smem 或 Xmem 值复制到 T 寄存器中。如果使用了 R 后缀执行凑整操作，则 dst 加 2^{15} 后低 16 位清零。指令受 OVM 和 FRCT 状态标志位的影响，执行结果影响 OVdst。

举例：

MPY 13,A



■ MPYA

语法：1. MPYA Smem

2. MPYA dst

说明：

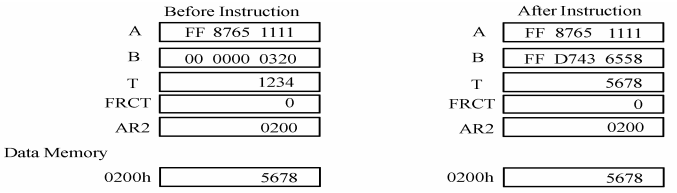
语法 1：累加器 A 高端（32~16 位）与 Smem 值相乘，结果放入累加器 B。Smem 值复制到 T 寄存器。

语法 2：累加器 A 高端（32~16 位）与 T 寄存器值相乘，结果放入 dst。

指令受 OVM 和 FRCT 状态标志位的影响，语法 1 中执行结果影响 OVB，语法 2 中执行结果影响 OVdst。

举例：

MPYA *AR2



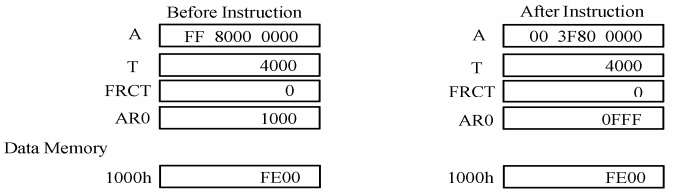
■ MPYU

语法：MPYU Smem, dst

说明：无符号 T 寄存器值与无符号 Smem 值相乘，结果存放于 dst。此时指令相当于两个最高位均为 0 的有符号 17×17 位乘法。该指令在多精度乘法中广泛应用。指令受 OVM 和 FRCT 状态标志位的影响，执行结果影响 OVdst。

举例：

MPYU *AR0-，A



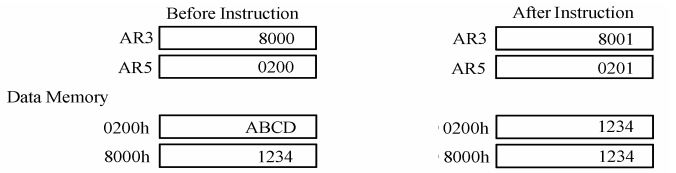
■ MVDD

语法：MVDD Xmem, Ymem

说明：将通过 Xmem 寻址的数据存储区单元的值复制到 Ymem 寻址的数据存储区单元中去。

举例：

MVDD *AR3+，*AR5+



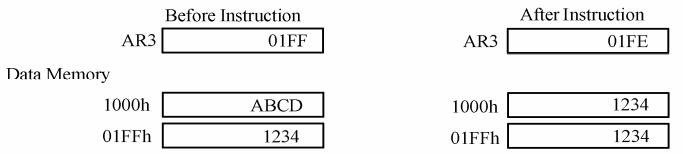
■　MVDK

语法：MVDK　Smem,　dmad

说明：将 Smem 值复制到一个通过 dmad 寻址的数据存储区单元。循环执行该指令可以将数据存储区的连续单元搬移到另一数据区。一旦循环流水启动，该指令就变成单周期指令。

举例：

MVDK *AR3-，1000h



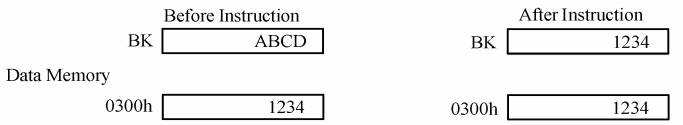
■　MVDM

语法：MVDM　dmad,　MMR

说明：将数据存储区单元 dmad 值复制到一个存储区映射寄存器 MMR 中。

举例：

MVDM 300h，BK



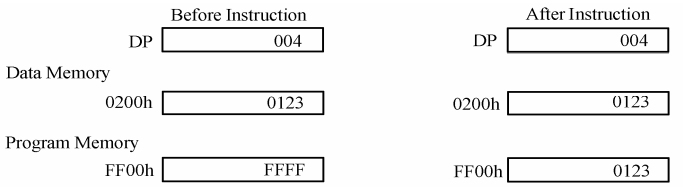
■　MVDP

语法：MVDP　Smem,　pmad

说明：将 Smem 值复制到由 pmad 寻址的程序存储区单元中。可以通过循环将数据存储区中的连续单元搬移到由 16 位立即数寻址的连续程序存储区空间。源和目的块不必全部在片内或片外。当循环流水启动时，该指令变成单周期指令。此外，循环执行该指令时，中断被禁止。

举例：

MVDP 0，0FE00h



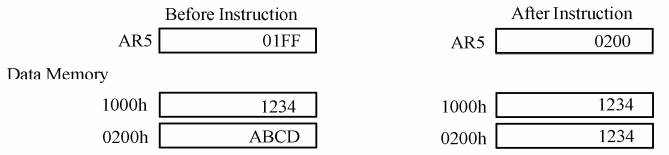
■　MVKD

语法：MVKD　dmad,　Smem

说明：将一个 dmad 寻址的数据存储区单元数据复制到 Smem 中。循环执行该指令可以将数据存储区的连续单元搬移到另一数据区。一旦循环流水启动，该指令就变成单周期指令。

举例：

MVKD 1000h，*+AR5



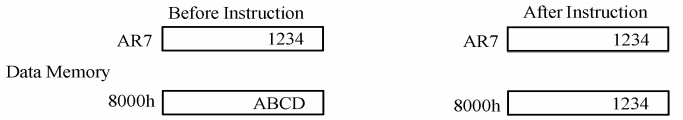
■ MVMD

语法：MVMD MMR, dmad

说明：将存储区映射寄存器 MMR 的数据转移到 dmad 寻址的数据存储区单元。一旦循环流水启动，该指令就变成了单周期指令。

举例：

MVMD AR7,8000h



■ MVMM

语法：MVMM MMRx, MMRy

说明：把存储区映射寄存器 MMRx 中的内容转移到另一个存储区映射寄存器 MMRy 中。MMRx 和 MMRy 只能为 9 种操作数：AR0~AR7 和 SP。

举例：

MVMM SP,AR1



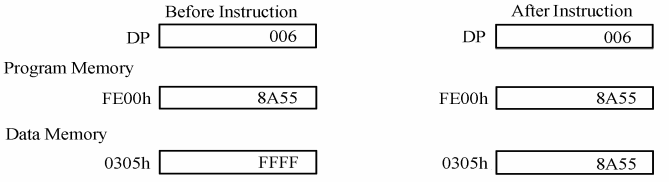
■ MVPD

语法：MVPD pmad, Smem

说明：将由 pmad 寻址的程序存储区值复制到 Smem。循环执行该指令可以将程序存储区中的连续单元搬移到数据存储区的连续单元中。一旦循环流水启动，该指令就变成了单周期指令。

举例：

MVPD 0FE00h,5



■ NEG

语法：NEG src [, dst]

说明：计算 src 的二进制补码，并把结果存放在 dst 中。若没有定义 dst，则存放在 src 中。如果累加器的值不为零，则指令将进位位 C 清为 0；如果累加器的值等于零，则指令将进位位 C 置 1。如果 src 值为 FF 8000 0000h，NEG 指令计算其补码时超过了累加器允许的最大值。此时若 OVM=1，则结果为 00 7FFF FFFFh；若 OVM=0，则结果为 00 8000 0000h。dst 的 OV 位在这两种情况下都设置为 1，指示结果溢出。

举例：

NEG A,B



■ NOP

语法：NOP

说明：该指令除了执行 PC 值加 1 以外不执行任何操作。该指令在执行延迟和解决流水线冲突方面常被采用。

■ NORM

语法：NORM　src　[, dst]

说明：将 src 中有符号数左移 TS 位，结果存放在 dst 中。若没有指定 dst，则存放在 src 中。该指令常与 EXP 指令结合使用，完成归一化操作。定点数的归一化指通过寻找符号扩展的高位，将定点数分为尾数和指数两部分，其中指数由 EXP 确定。首先 EXP 指令确定移位位数并存放在 T 寄存器中，然后使用 NORM 指令在单周期内对累加器内容进行移位，实现归一化。指令受 OVM 和 SXM 状态标志位的影响，执行结果影响 OVdst。

举例：

NORM B,A



■ OR

语法：1．OR　Smem,　src

2．OR　#lk　[, SHFT],　src　[, dst]

3．OR　#lk,　16,　src　[, dst]

4．OR　src　[, SHIFT]　[, dst]

说明：将 src 与 Smem、左移后的 16 位立即数、dst 或者 src 自身移位后的值进行或运算，结果存放在 dst 中。若没有指定 dst，则存放在 src 中。对于指令中的左移，低位填零高位不进行符号扩展；右移时高位不进行符号扩展。对于语法 1，指令执行后 src 的 39~16 位不变。

举例：

OR　A，+3，B



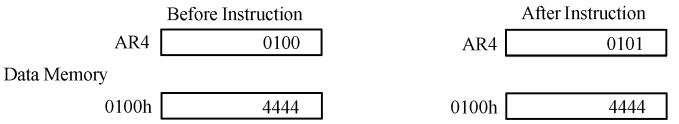
■ ORM

语法：ORM　#lk,　Smem

说明：将 Smem 值与一个 16 位立即数进行或运算，结果存放在 Smem 中。该指令不能循环执行。

举例：

ORM 0404h，*AR4+



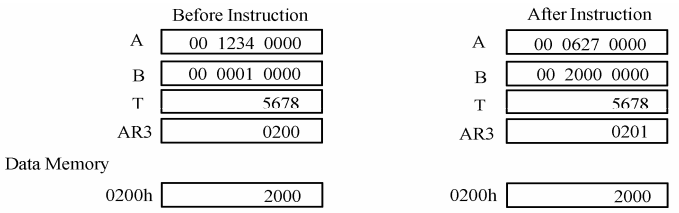
■ POLY

语法：POLY　Smem

说明：Smem 值左移 16 位后存放在累加器 B 中。同时累加器 A 的高端（32~16 位）与 T 寄存器的值相乘，乘积与累加器 B 相加后再舍入，最终结果存放在累加器 A 中。该指令在多项式计算中经常使用。指令受 FRCT，OVM 和 SXM 状态标志位的影响，执行结果影响 OVA。

举例：

POLY *AR3+%；（FRCT=0）



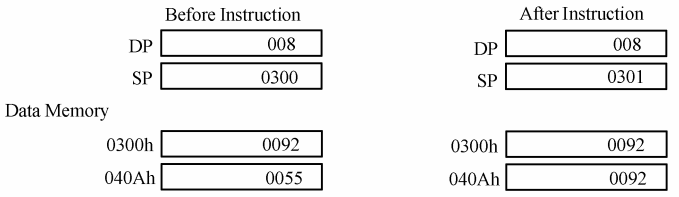
■ POPD

语法：POPD　Smem

说明：将由 SP 寻址的数据存储区单元的内容复制到由 Smem 确定的数据存储区单元中。最后 SP 指针加 1。

举例：

POPD 10



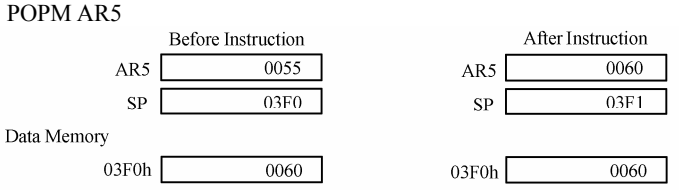
■ POPM

语法：POPM　MMR

说明：将由 SP 寻址的数据存储区单元的内容复制到存储区映射寄存器 MMR 中。最后 SP 指针加 1。

举例：

POPM AR5



■ PORTR

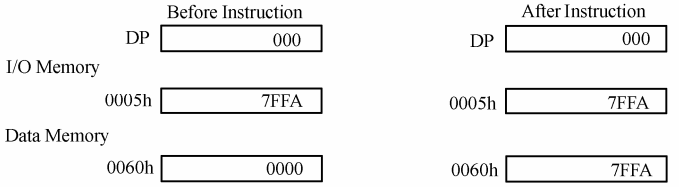
语法：PORTR　PA,　Smem

说明：从外部 I/O 口 PA（16 位立即数地址）将一个 16 位数读入到 Smem 中。$\overline{IS}$ 信号变

为低电平表明在访问 I/O 口；$\overline{\text{IOSTRB}}$ 和 READY 的时序与读外部数据存储区的时序相同。

举例：

```
PORTR 05，INDAT；INDAT .EQU 60h
```



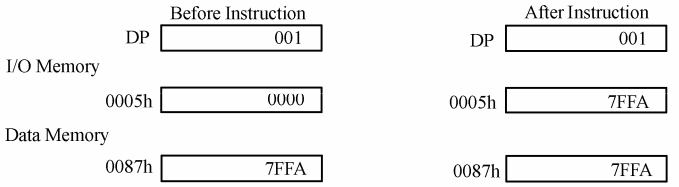
■　PORTW

语法：PORTW　Smem,　PA

说明：将 16 位 Smem 值写到外部 I/O 口 PA 中。$\overline{\text{IS}}$ 信号变为低电平表明在访问 I/O 口；$\overline{\text{IOSTRB}}$ 和 READY 的时序与读外部数据存储区的时序相同。

举例：

```
PORTW OUTDAT，5h；OUTDAT .equ 07h
```



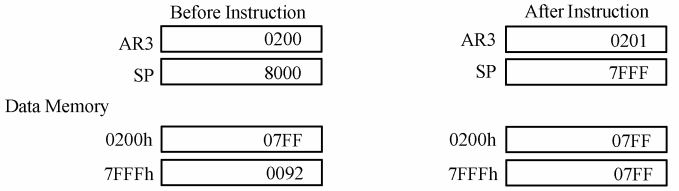
■　PSHD

语法：PSHD　Smem

说明：堆栈指针 SP 减 1，Smem 值压入 SP 指向的数据存储区单元中。

举例：

```
PSHD *AR3+
```



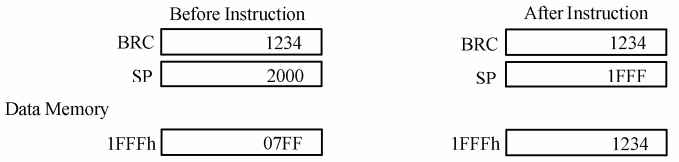
■　PSHM

语法：PSHM　MMR

说明：堆栈指针 SP 减 1，存储区映射寄存器 MMR 内容压入 SP 所指向的数据存储区单元中。

举例：

```
PSHM BRC
```



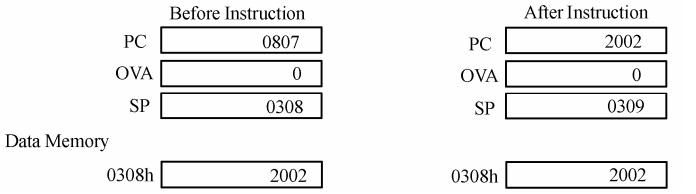
■　RC[D]

语法：RC[D]　cond　[, cond　[, cond]]

说明：如果满足给定的条件，堆放在栈顶的数据存储区值弹出到 PC 中，堆栈指针 SP 加 1，从而实现返回。如果不满足条件则仅执行 PC 加 1 的操作。若为延迟方式，此时紧跟该指令的两条单字指令或 1 条双字指令先被取出执行，且它们不会影响被测试的条件。该指令可以对多个条件进行测试，表 7-3 和表 7-4 列出了各种条件及其分组分类情况。

举例：

RC AGEQ, ANOV



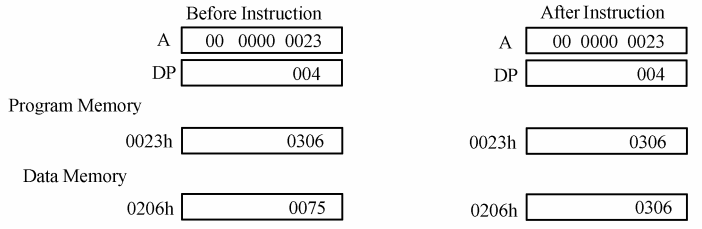
■ READA

语法：READA　Smem

说明：将由累加器 A 所确定的程序存储区单元中的值传送给 Smem。如果循环执行该指令，则可以把一个连续单元（由累加器 A 确定起始地址）搬移到一个连续的使用间接寻址的数据存储区空间中去。

举例：

READA 6



■ RESET

语法：RESET

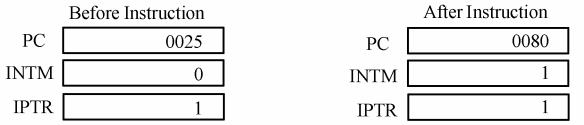
说明：指令实现非屏蔽的 PMST、ST0 和 ST1 复位，重新赋予默认值。这些寄存器中各个状态位的赋值情况如下：

（IPTR）<<7→PC　0→OVA　0→OVB　1→C　1→TC
0→ARP　0→DP　1→SXM　0→ASM　0→BRAF
0→HM　1→XF　0→C16　0→FRCT　0→CMPT
0→CPL　1→INTM　0→IFR　0→OVM

该指令不受 INTM 指令的影响，但它对 INTM 置位以禁止中断。该指令不能循环执行。

举例：

RESET



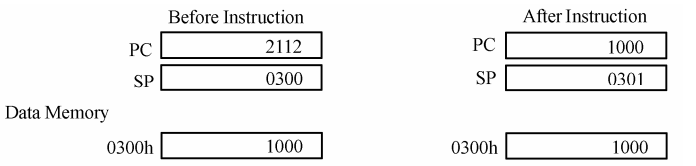
■ RET[D]

语法：RET[D]

说明：将栈顶的 16 位数据弹出到 PC 中，从这个地址继续执行，堆栈指针 SP 加 1。如果是延迟返回，则紧跟该指令的两条单字指令或 1 条双字指令先被取出执行。该指令不能循环执行。

举例：

RET

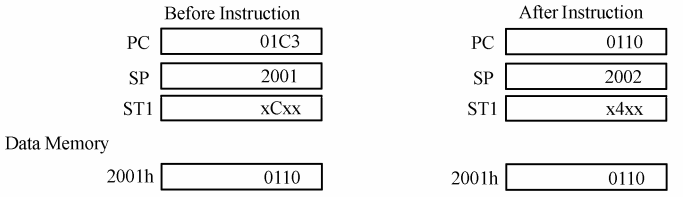


■ RETE[D]

语法：将栈顶的 16 位数据弹出到 PC 中，从这个地址继续执行，堆栈指针 SP 加 1。指令同时将 ST1 中的 INTM 自动清零，允许产生中断。如果是延迟返回，则紧跟该指令的两条单字指令或 1 条双字指令先被取出执行。该指令不能循环执行。

举例：

RETE

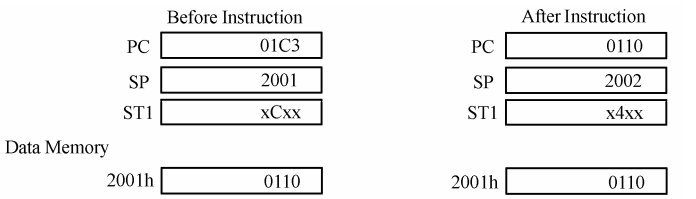


■ RETF[D]

语法：将 RTN 寄存器的内容赋给 PC，然后 SP 加 1。RTN 中保存着中断服务子程序的返回地址，因此该指令的返回地址是在返回时得到的而不是从堆栈顶部取出的，这样可以实现快速返回。同时指令自动对 INTM 清零，允许中断。如果是延迟返回，则紧跟该指令的两条单字指令或 1 条双字指令先被取出执行。注意只有在该中断服务子程序执行期间没有调用且没有执行其他中断子程序时才能使用该指令。

举例：

RETF



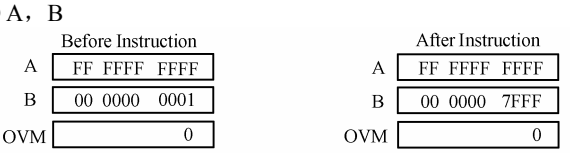
■ RND

语法：RND　src　[, dst]

说明：将 src 内容与 2^{15} 相加以进行舍入，结果放入 dst 中。若未定义 dst，则结果存放于 src 中。该指令不能被循环执行。指令受 OVM 状态标志位的影响。

举例：

RND A，B



■ ROL

语法：ROL　src

说明：src 循环左移 1 位，进位位 C 的值移入 src 的最低位，src 的最高位移入 C 中。保护位清零。

举例：

ROL A



■ ROLTC

语法：ROLTC　src

说明：src 循环左移 1 位，TC 值移入 src 的最低位，src 的最高位移入 C 中。保护位清零。

举例：

ROLTC A



■ ROR

语法：ROR　src

说明：src 循环右移 1 位，进位位 C 移入 src 的最高位，src 的最低位移入 C 中。保护位清零。

举例：

ROR A



■ RPT

语法：1．RPT　Smem

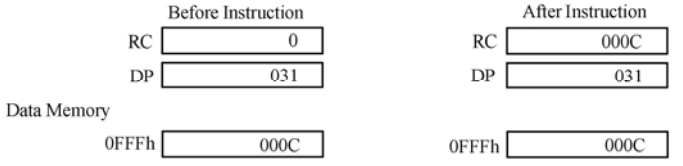
2．RPT　#K

3．RPT　#lk

说明：单指令循环。循环次数 n 由 Smem、常数 K 或 lk 确定，并放入循环计数器 RC 中。这样紧接着的下一条指令将循环执行 n+1 次。该指令不能被循环执行，也就是说不能循环嵌套。

举例：

RPT DAT127 ； DAT127 .EQU 0FFF



■ RPTB[D]

语法：RPTB[D]　pmad

说明：块循环指令。循环次数由块循环计数器 BRC 确定，BRC 必须在指令执行前被装

入。执行指令时，块循环起始寄存器 RSA 装入 PC+2（若有 D 后缀时为 PC+4）；块循环尾地址寄存器 REA 中装入 pmad。单指令循环（RPT）也属于块循环。块循环在执行过程中可以被中断，为了保证循环能够正确执行，中断时必须要保存 BRC、RSA 和 REA 寄存器且正确设置块循环有效标志 BRAF。如果是延迟方式，则紧跟该指令的两条单字指令或 1 条双字指令先被取出执行。注意块循环可以通过将 BRAF 清零来终止，并且该指令不能循环执行。指令执行结果影响 BRAF。

举例：

```
STM     #99, BRC
NOP
                RPTB    end_block – 1     ;end_block 为循环块的底部
```

Before Instruction		After Instruction	
PC	1000	PC	1002
BRC	1234	BRC	0063
RSA	5678	RSA	1002
REA	9ABC	REA	end block-1

■ RPTZ

语法：RPTZ　dst,　#lk

说明：dst 清零且执行下一条指令 lk+1 次，其中常数 lk 存放在循环计数器 RC 中。

举例：

```
RPTZ A, 1023 ; 重复执行下一条指令 1024 次
             STL A, *AR2+
```

Before Instruction		After Instruction	
A	0F FE00 8000	A	00 0000 0000
RC	0000	RC	03FF

■ RSBX

语法：RSBX　N,　SBIT

说明：对状态寄存器 ST0 和 ST1 的指定标志位清零。N 指示为哪一个状态寄存器，SBIT 指示状态寄存器的哪一位被修改。还有一种方便的方法是直接用状态寄存器的域名作为操作数，此时就不需要 N 和 SBIT。该指令不能循环执行。

举例 1：

```
RSBX SXM    ; SXM 指示 n=1 且 SBIT=8
```

Before Instruction		After Instruction	
ST1	35CD	ST1	34CD

举例 2：

```
RSBX 1，8
```

Before Instruction		After Instruction	
ST1	35CD	ST1	34CD

■ SACCD

语法：SACCD　src,　Xmem,　cond

说明：如果满足表 7-7 条件，源累加器 src 左移（ASM-16）位后存放到 Xmem 指定的存储区单元中。若不满足条件，则 Xmem 值保持不变。

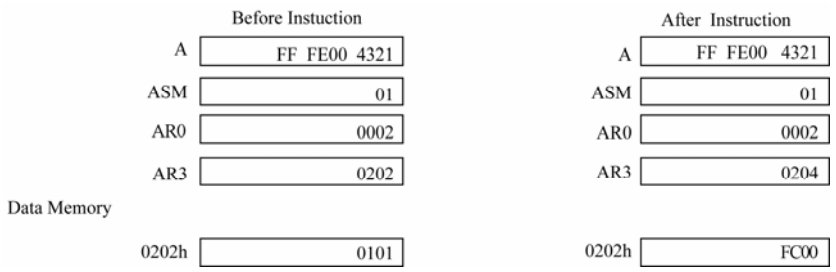
指令受 ASM 和 SXM 状态标志位的影响。

举例：

表 7-7　指令测试条件

条 件	说 明	条 件	说 明
AEG	（A）=0	BAEG	（B）=0
ANEG	（A）≠0	BNEG	（B）≠0
AGT	（A）>0	BGT	（A）>0
AGEQ	（A）≥0	BGEQ	（B）≥0
ALT	（A）<0	BLT	（B）<0
ALEQ	（A）≤0	BLEQ	（B）≤0

SACCD A，　*AR3+0%，ALT



■　SAT

语法：SAT　src

说明：不管 OVM 如何取值，指令都将 src 的值饱和为 32 位。指令执行结果影响 OVsrc。

举例 1：

SAT B



举例 2：

SAT A



举例 3：

SAT B



■　SFTA

语法：SFTA　src,　SHIFT　[, dst]

说明：对 src 进行算术移位，结果存放在 dst 中。若没有指定 dst，则存放在 src 中。指令的执行方式由 SHIFT 决定：

SHIFT<0，则 src 的第（–SHIFT–1 ）位复制到进位位 C 中，40 位 src 右移 SHIFT 位，结果存入 dst。此时若 SXM=1，要进行符号扩展，dst 的高 SHIFT 的各位写入 src 最高位的值（如果没有指定 dst，则 src 最高位的值写入 src 的高 SHIFT 位）。反之，若 SXM=0，不进行符号扩展，将 0 写入 dst 高 SHIFT 的各位（如果没有指定 dst，则将 0 写入 src 的高 SHIFT 位）。

若 SHIFT>0，则 src 的第（ 39–SHIFT +1）位复制到进位位 C 中。40 位 src 左移 SHIFT 位，然后将 0 写入 dst 的低 SHIFT 位（如果没有指定 dst，则将 0 写入 src 的低 SHIFT 位）。

指令受 OVM 和 SXM 状态标志位的影响，执行结果影响 C 和 OVdst（若未指定 dst，则为 OVsrc）。

举例：

SFTA A，-5，B

	Before Instruction	After Instruction
A	FF 8765 0055	FF 8765 0055
B	00 4321 1234	FF FC3B 2802
C	x	1
SXM	1	1

■ SFTC

语法：SFTC　src

说明：若 src 有两个有效的符号位，则 32 位 src 左移 1 位，保护位保持不变，同时测试控制位 TC=0；若只有一个符号位，则 TC=1。

举例：

SFTC　A

	Before Instuction	After Instruction
A	FF FFFF F001	FF FFFF E002
TC	X	0

■ SFTL

语法：SFTL　src,　SHIFT　[, dst]

说明：对 src 进行逻辑移位，结果存放在 dst 中。若没有指定 dst，则存放在 src 中。指令的执行方式由 SHIFT 决定。

若 SHIFT<0，则：

① src 的第（–SHIFT–1）位复制到进位位 C 中；

② src 低 32 位逻辑右移 SHIFT 位，结果存放到 dst；

③ dst 的第 39 位至第（ 31+（SHIFT+1））位填零。

若 SHIFT=0，则进位位 C=0。

若 SHIFT>0，则：

① src 的第（31–（SHIFT–1））位复制到进位位 C 中；

② src 的第（31– SHIFT）位至第 0 位逻辑左移 SHIFT 位，结果存放到 dst；

③ dst 的低 SHIFT 位和高 8 位填零，若没有指定 dst，则 src 的低 SHIFT 位和高 8 位填零。

指令执行结果影响 C。

举例：

SFTL A，-5，B

	Before Instruction	After Instruction
A	FF 8765 0055	FF 8765 0055
B	FF 8000 000	00 043B 2802
C	0	1

■ SQDST

语法：SQDST　Xmem,　Ymem

说明：常在单指令循环中用于计算两个向量距离的平方。执行过程为：累加器 A 的高端（32~16 位）平方后加到累加器 B 中。同时，Xmem 与 Ymem 相减的差值左移 16 位存放在累加器 A 中。指令受 OVM，FRCT 和 SXM 状态标志位的影响，执行结果影响 C，INTM，OVA 和 OVB。

举例：

SQDST *AR3+，AR4+

	Before Instruction	After Instruction
A	FF ABCD 0000	FF FFAB 0000
B	00 0000 0000	00 1BB1 8229
FRCT	0	0
AR3	0100	0101
AR4	0200	0201
Data Memory		
0100h	0055	0055
0200h	00AA	00AA

■ SQUR

语法：1. SQUR Smem, dst

2. SQUR A, dst

说明：将 Smem 值或累加器 A 的高端（32~16 位）平方后，结果存放在 dst 中。如果操作数为 Smem，同时还要将 Smem 值存入 T 寄存器。如果为累加器 A，则 T 不受影响。指令受 OVM 和 FRCT 状态标志位的影响，执行结果影响 OVsrc。

举例：

SQUR 30，B

	Before Instruction	After Instruction
B	00 0000 01F4	00 0000 00E1
T	0003	000F
FRCT	0	0
DP	006	006
Data Memory		
031Eh	000F	000F

■ SQURA

语法：SQURA Smem, src

说明：将 Smem 值复制到 T 寄存器；然后 Smem 的平方与 src 相加，结果存放在 src 中。指令受 OVM 和 FRCT 状态标志位的影响，执行结果影响 OVsrc。

举例：

SQURA 30，B

	Before Instruction	After Instruction
B	00 0320 0000	00 0320 00E1
T	0003	000F
FRCT	0	0
DP	006	006
Data Memory		
031Eh	000F	000F

■ SQURS

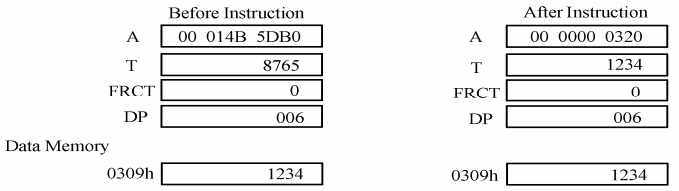
语法：SQURS Smem, src

说明：将 Smem 值复制到 T 寄存器；然后 src 与 Smem 的平方相减，结果存放在 src 中。

指令受 OVM 和 FRCT 状态标志位的影响，执行结果影响 OVsrc。

举例：

SQURS 9，A



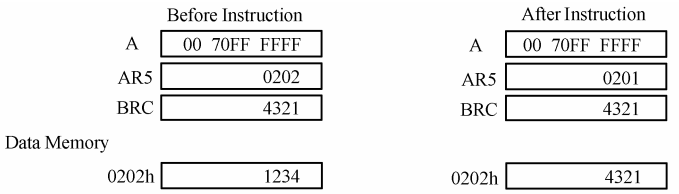
■　SRCCD

语法：SRCCD　Xmem,　cond

说明：如果满足条件（测试条件见表 7-7），则将块循环计数器 BRC 中的内容存放到 Xmem 中；反之，则保持 Xmem 不变。

举例：

SRCCD *AR5-，AGT



■　SSBX

语法：SSBX　N,　SBIT

说明：将状态寄存器 ST0 和 ST1 的指定标志位置 1。N 指示为哪一个状态寄存器，SBIT 指示状态寄存器的哪一位被修改。还有一种方便的方法是直接用状态寄存器的域名作为操作数，此时就不需要 N 和 SBIT。该指令不能循环执行。

举例 1：

SSBX　SXM　　;SXM 指示 n=1 且 SBIT=8



举例 2：

SSBX 1,8



■　ST

语法：1．ST　T,　Smem

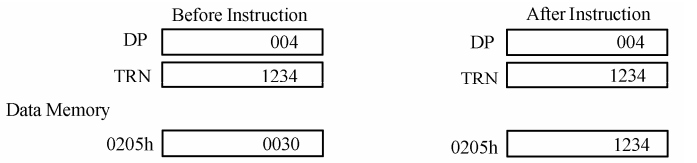
2．ST　TRN,　Smem

3．ST　#lk,　Smem

说明：将 T 寄存器的内容、TRN 寄存器的内容或一个 16 位常数 lk 存放到 Smem 中。

举例：

ST TRN，5



■ STH

语法：1．STH　src,　Smem

2．STH　src,　ASM,　Smem

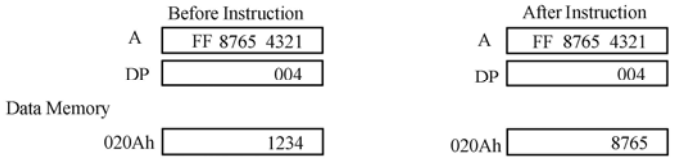
3．STH　src,　SHFT,　Xmem

4．STH　src　[, SHIFT],　Smem

说明：将 src 的高端（31~16 位）存放到数据存储单元中去。若 src 需要移位，则左移位数由（ASM-16）、（SHFT-16）或（SHIFT-16）确定。移位后 src 的 31~16 位存放到 Smem 或 Xmem 中。指令受 SXM 状态标志位的影响。

举例：

STH　A，10



■ STL

语法：1．STL　src,　Smem

2．STL　src,　ASM,　Smem

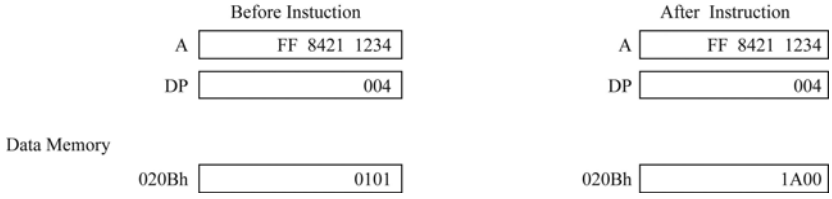
3．STL　src,　SHFT,　Xmem

4．STL　src　[, SHIFT],　Smem

说明：将 src 的低端（15~0 位）存放到数据存储单元中去。若 src 需要移位，则左移位数由 ASM、SHFT 或 SHIFT 确定。移位后 src 的 15~0 位存放到 Smem 或 Xmem 中。当移位数为正数时，低位填零。指令受 SXM 状态标志位的影响。

举例：

STL　A，7，11



■ STLM

语法：STLM　src,　MMR

说明：将累加器 src 的低端（15~0 位）存放到 MMR 中。不管页指针 DP 或 ARx 的高 9 位为何值，有效地址的高 9 位都清零，这样无须修改 ST0 中的 DP 域，src 低端就可以存放在数据页第 0 页的任何一个存储单元。

举例：

STLM　A，BRC



■ STM

语法：STM　#lk，　MMR

说明：将 16 位常数 lk 存放到 MMR 或数据页第 0 页的一个存储单元中。不管页指针 DP 或 ARx 的高 9 位为何值，有效地址的高 9 位都清零。

举例：

```
STM　0FFFFh，IMR
```



■ ST || ADD

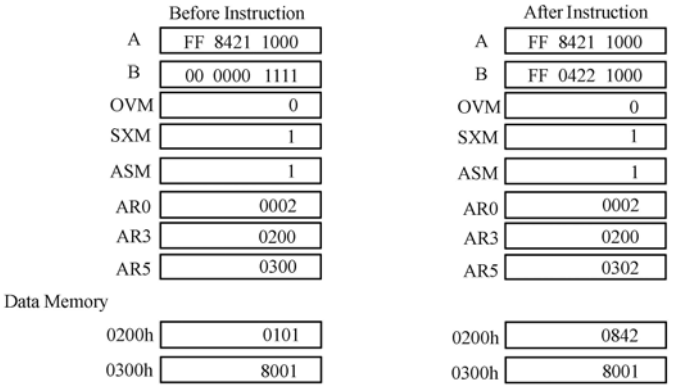
语法：　ST　src,　Ymem

　　|| ADD　Xmem,　dst

说明：src 左移（ASM-16）位后存放到 Ymem 中；同时并行执行 dst_的内容与左移 16 位后的 Xmem 相加，结果存放在 dst 中。若 src 与 dst 为同一累加器，则存放到 Ymem 中的值为 src 修改前的值。指令受 OVM，ASM 和 SXM 状态标志位的影响，执行结果影响 C 和 OVdst。

举例：

```
ST　A，*AR3
||ADD *AR5+0%，B
```



■ ST || LD

语法：1.　ST　src,　Ymem

　　|| LD　Xmem,　dst

　　2.　ST　src,　Ymem

　　|| LD　Xmem,　T

说明：src 左移（ASM-16）位后存放到 Ymem 中；同时并行执行 Xmem 左移 16 位后装入 dst 或 Xmem 直接装入 T 寄存器中。若 src 与 dst 为同一累加器，则存放到 Ymem 中的值为 src 修改前的值。指令受 OVM 和 ASM 状态标志位的影响，执行结果影响 C。

举例：

```
ST B，*AR2-
||LD *AR4+，A
```

	Before Instruction	After Instruction
A	00 0000 001C	FF 8001 0000
B	FF 8421 1234	FF 8421 1234
SXM	1	1
ASM	1C	1C
AR2	01FF	01FE
AR4	0200	0201
Data Memory		
01FFh	xxxx	F842
0200h	8001	8001

■ ST || MAC[R]

语法：ST src, Ymem

|| MAC[R] Xmem, dst

说明：src 左移（ASM-16）位后存放到 Ymem 中；同时并行执行 T 寄存器内容与 Xmem 相乘，乘积与 dst 相加后存放在 dst 中。若 src 与 dst 为同一累加器，则存放到 Ymem 中的值为 src 修改前的值。如果使用了 R 后缀，则乘累加结果要进行舍入操作，即 dst 加 2^{15} 后低端（15~0 位）清零。指令受 OVM，FRCT，ASM 和 SXM 状态标志位的影响，执行结果影响 C 和 OVdst。

举例：

```
ST A，*AR4-
||MAC *AR5，B
```

	Before Instruction	After Instruction
A	00 0011 1111	00 0011 1111
B	00 0000 1111	00 010C 9511
T	0400	0400
ASM	5	5
FRCT	0	0
AR4	0100	00FF
AR5	0200	0200
Data Memory		
100h	1234	0222
200h	4321	4321

■ ST || MAS[R]

语法：ST src, Ymem

|| MAS[R] Xmem, dst

说明：src 左移（ASM-16）位后存放到 Ymem 中；同时并行执行 T 寄存器内容与 Xmem 相乘，dst 内容与该乘积相减后存放在 dst 中。若 src 与 dst 为同一累加器，则存放到 Ymem 中的值为 src 修改前的值。如果使用了 R 后缀，则乘累加结果要进行舍入操作，即 dst 加 2^{15} 后低端（15~0 位）清零。指令受 OVM，FRCT，ASM 和 SXM 状态标志位的影响，执行结果影响 C 和 OVdst。

举例：

```
ST A，*AR4+
||MAS *AR5，B
```

	Before Instruction	After Instruction
A	00 0011 1111	00 0011 1111
B	00 0000 1111	FF FEF3 8D11
T	0400	0400
ASM	5	5
FRCT	0	0
AR4	0100	0101
AR5	0200	0200
Data Memory		
0100h	1234	0222
0200h	4321	4321

■　ST || MPY

语法：ST　src,　Ymem

　　|| MPY　Xmem,　dst

说明：src 左移（ASM-16）位后存放到 Ymem 中；同时并行执行 T 寄存器内容与 Xmem 相乘，乘积放在 dst 中。若 src 与 dst 为同一累加器，则存放到 Ymem 中的值为 src 修改前的值。指令受 OVM，FRCT，ASM 和 SXM 状态标志位的影响，执行结果影响 C 和 OVdst。

举例：

```
ST A，*AR3+
||MPY *AR5+，B
```

	Before Instruction	After Instruction
A	FF 8421 1234	FF 8421 1234
B	xx xxxx xxxx	00 2000 0000
T	4000	4000
ASM	00	00
FRCT	1	1
AR3	0200	0201
AR5	0300	0301
Data Memory		
0200h	1111	8421
0300h	4000	4000

■　ST || SUB

语法：ST　src,　Ymem

　　|| SUB　Xmem,　dst

说明：src 左移（ASM-16）位后存放到 Ymem 中；同时并行执行 Xmem 左移 16 位后与 dst 相减，结果存放在 dst 中。若 src 与 dst 为同一累加器，则存放到 Ymem 中的值为 src 修改前的值。指令受 OVM，ASM 和 SXM 状态标志位的影响，执行结果影响 C 和 OVdst。

举例：

```
ST A，*AR3-
||SUB *AR5+0%，B
```

	Before Instruction	After Instruction
A	FF 8421 0000	FF 8421 0000
B	00 1000 0001	FF FBE0 0000
ASM	01	01
SXM	1	1
AR0	0002	0002
AR3	01FF	01FE
AR5	0300	0302
Data Memory		
01FFh	1111	0842
0300h	8001	8001

■ STRCD

语法：STRCD Xmem, cond

说明：如果满足表 7-7 的条件，则将 T 寄存器内容存放到 Xmem 中；反之，Xmem 保持不变。

举例：

STRCD *AR5-，AGT

	Before Instruction	After Instruction
A	00 70FF FFFF	00 70FF FFFF
T	4321	4321
AR5	0202	0201
Data Memory		
0202h	1234	4321

■ SUB

语法：1．SUB Smem, src
2．SUB Smem, TS, src
3．SUB Smem, 16, src [, dst]
4．SUB Smem [, SHIFT], src [, dst]
5．SUB Xmem, SHFT, src
6．SUB Xmem, Ymem, dst
7．SUB #lk [, SHFT], src [, dst]
8．SUB #lk, 16, src [, dst]
9．SUB src [, SHIFT] [, dst]
10．SUB src, ASM [, dst]

说明：将指定累加器中或存储器单元中的数据作为被减数，减去一个 16 位减数，这个减数为 Smem、Xmem、16 位立即数 lk 或 src 经移位后的值。如果指定了 dst，则减法结果存放在 dst 中；否则结果存放于 src 中。注意语法 6 比较特殊，它的执行过程为：Xmem 和 Ymem 分别左移 16 位后再相减，结果存入 dst 中。第 9、10 条的执行过程为：dst 减去 src 移位后的值，结果存入 dst。指令受 OVM 和 SXM 状态标志位的影响，执行结果影响 C 和 OVdst（若未指定 dst，则为 OVsrc）。

举例：

SUB *AR1+，14，A

	Before Instruction	After Instruction
A	00 0000 1200	FF FAC0 1200
C	x	0
SXM	1	1
AR1	0100	0101
Data Memory		
0100h	1500	1500

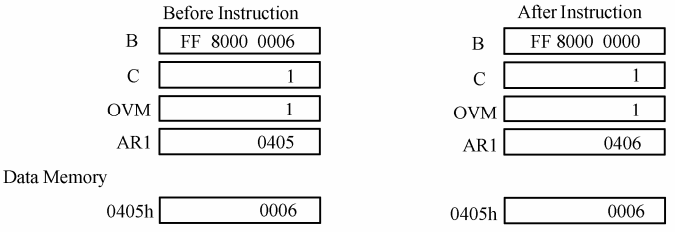
■　SUBB

语法：SUBB　Smem,　src

说明：从 src 中减去 Smem 的值和进位位 C 的逻辑反，结果不进行符号扩展并放入 src 中。指令受 OVM 和 C 状态标志位的影响，执行结果影响 C 和 OVsrc。

举例：

SUBB *AR1+，B



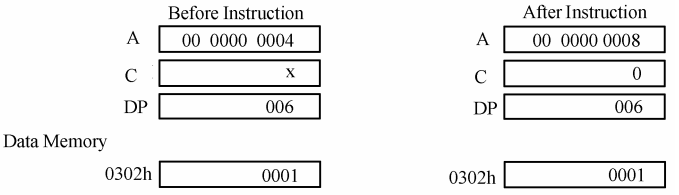
■　SUBC

语法：SUBC　Smem,　src

说明：src 减去 Smem 左移 15 位后的值，若结果大于或等于 0，则结果左移 1 位再加 1，最终结果存放到 src 中，否则 src 左移 1 位并存入 src 中。该指令在除法运算（用移位减实现除法）中经常被使用，此时被除数必须初始化为正数（src 的 31 位为 0），且移位后也必须保持为正。除数受 SXM 影响。在进行除法运算时，须假定除数与被除数均为正数，若 SXM=1，则除数的最高位必须为 0；若 SXM=0，则除数可以为任何一个 16 位数。注意：当 src 产生溢出时不进行饱和运算。执行结果影响 C 和 OVsrc。

举例：

SUBC 2，A



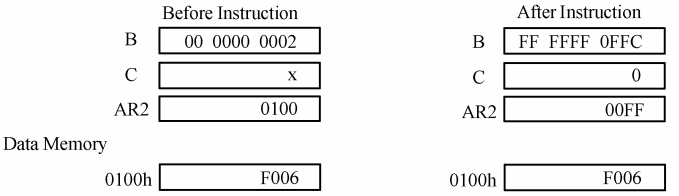
■　SUBS

语法：SUBS　Smem,　src

说明：src 减去 Smem 值，结果存放到 src 中。无论 SXM 取值多少，Smem 都被视为 16 位无符号数。指令受 OVM 状态标志位的影响，执行结果影响 C 和 OVsrc。

举例：

SUBS *AR2-，B



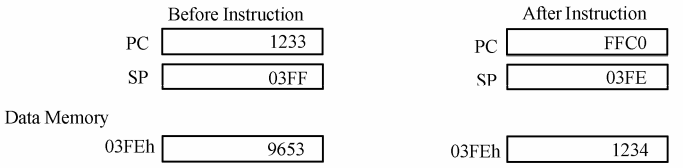
■　TRAP

语法：TRAP K

说明：将返回地址（PC＋1）压入栈顶后，PC 指向由 K 指定的中断向量。这样用户可以通过该指令来执行任何中断服务子程序。该指令是非屏蔽的，不受 INTM 影响也不会影响 INTM。该指令不能循环执行。

举例：

TRAP 10h



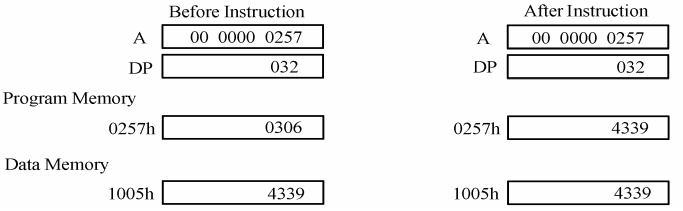
■ WRITA

语法：WRITA Smem

说明：将 Smem 值传送到由累加器 A 确定的程序存储区单元。通过循环指令可以将数据存储区的连续块（采用间接寻址）转移到由 PAR 寻址的程序存储区的连续块中。PAR 的初始值由累加器 A 确定。一旦循环流水启动，该指令就变成单周期指令。

举例：

WRITA 5



■ XC

语法：XC n, cond [, cond [, cond]]

说明：指令中 n=1 或 2。如果满足表 7-3 和表 7-4 列出的条件，此时若 n=1，则执行紧接着该指令的一条单字指令；若满足条件且 n=2，则执行紧接着该指令的一条双字指令或两条单字指令。如果不满足条件，则执行 nop 操作，执行的次数由 n 确定。注意该指令和紧接着的两个指令字执行时是不能被中断的。由于被测试条件是在指令执行之前的两个周期被采样，因此即使该指令前的两条单字指令或一条双字指令修改了条件也不会影响指令的执行。

举例：

```
XC    1, ALEQ
MAR   *AR1+
ADD   A, DAT100
```



■ XOR

语法：1. XOR Smem, src

2．XOR #lk [, SHFT], src [, dst]

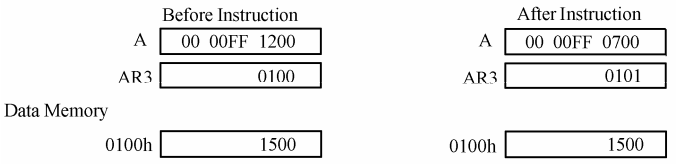
3．XOR #lk, 16, src [, dst]

4．XOR src [, SHIFT] [, dst]

说明：Smem 或 16 位常数按指令指定位数移位后与 src 执行异或操作，结果存放在 dst 中。若没有指定 dst，则存放在 src 中。左移时，低位填零，高位不进行符号扩展；右移时，不进行符号扩展。

举例：

XOR *AR3+，A



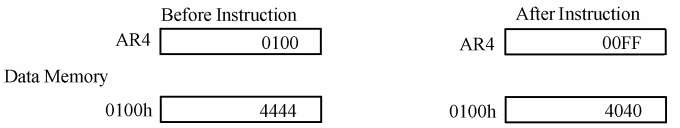
■　XORM

语法：XORM　#lk,　Smem

说明：Smem 值与 16 位常数执行异或操作，结果写到 Smem 中。该指令不能循环执行。

举例：

XORM　0404h，*AR4-



7.4　寻址方式

TMS320C5000 DSP 芯片的寻址方式分为数据寻址和程序寻址两大类。由于程序代码和数据都是存放在 DSP 的存储区中，因此运行程序时必然要对程序存储区以及数据存储区的相应单元进行访问，这样才能准确取出指令和操作数并可靠地运行。DSP 芯片提供了灵活多样的寻址方式，保证了程序的运行更加迅速有效。

7.4.1　数据寻址

TMS320C5000 提供了 7 种基本的数据寻址方式：

- 立即数寻址：指令中包含有一个固定的数；
- 绝对地址寻址：指令中包含一个固定的地址；
- 累加器寻址：将累加器的内容作为地址去访问程序存储区单元；
- 直接寻址：数据存储区地址由基地址（DP 或 SP）和偏移地址组成；数据存储区地址的高 9 位由基地址提供，低 7 位由偏移地址提供；
- 间接寻址：将辅助寄存器中的内容作为地址去访问存储区；
- 存储区映射寄存器寻址：修改映射寄存器的内容而不影响当前的 DP 或 SP 值；
- 堆栈寻址：将数据压入或者弹出系统堆栈。

下面按照顺序对这些寻址方式逐一介绍。

1．立即数寻址

立即数寻址方式中，由于操作数直接包含在指令当中，因此没有寻找数据地址的过程。操作数有两种类型：短立即数和长立即数。不同的指令类型需要不同长度的立即数。短立即

数长度为 3、5、8 或 9 位，可以放在一个字长的指令中。而长立即数为 16 位，放在两个字长的指令中。表 7-8 列出了可以包含立即数的指令，并指出了立即数的位数。

表 7-8　支持立即数的指令

3 位和 5 位立即数	8 位立即数	9 位立即数	16 位立即数
LD	FRAME LD RPT	LD	ADD　ORM　ADDM　RPT AND　RPTZ　ANDM　ST BITF　STM　CMPM　SUB LD　XOR　OR

注意：立即数寻址语句要在数值或符号前面加一个“#”号来说明它是一个立即数，否则会被认为是：一个地址。例如，要将数值 80h 装入累加器 A 中，指令应该表示为：

```
LD　#80h,　A
```

但若表示为：

```
LD　80h,　A
```

则指令变成了将数据存储区 80h 单元中的数据放入累加器 A 中。

下面用 RPT 指令说明一个立即数是如何放置在指令代码中的。若立即数为 8 位，则 RPT 为单字长指令，此时操作码放在指令的高半段（15~8 位），立即数放在指令的低半段（7~0 位），如图 7-1（a）所示。若立即数为 16 位，则 RPT 为双字长指令，此时操作码放在高位字中，立即数放在低位字中，如图 7-1（b）所示。



图 7-1　带立即数的 RPT 指令

2．绝对地址寻址

C54x 的绝对地址寻址有 4 种类型。

（1）数据存储区地址（dmad）寻址，主要指令有：

```
MVDK　Smem,　dmad
MVDM　dmad,　MMR
MVKD　dmad,　Smem
MVMD　MMR,　dmad
```

dmad 寻址的共同特点是：指令中的一个操作数用符号或者常数来指明其所在的数据空间地址。例如，要将 AR3 指向的数据存储区单元内容复制到数据空间地址为 SAMPLE 的单元中去，可以使用下面的语句：

```
MVDK　*AR3,　SAMPLE
```

其中，SAMPLE 指示的地址就是数据存储区地址，SAMPLE 也可以用常数表示的地址取代。例如，若 SAMPLE 表示的数据地址为 1000h，则上面的语句还可以表示为：

```
MVDK   *AR3,   1000h
```

（2）程序存储区地址（pmad）寻址，主要指令有：

```
FIRS   Xmem,   Ymem,   pmad
MACD   Smem,   pmad,   src
MACP   Smem,   pmad,   src
MVDP   Smem,   pmad
MVPD   pmad,   Smem
```

pmad 寻址的共同特点是：指令中的一个操作数用符号或者常数来指明其所在的程序空间地址。例如，要将 AR7 指向的数据存储区单元内容复制到程序空间地址为 TABLE 的单元中去，可以使用下面的语句：

```
MVDP   *AR7, TABLE
```

其中，TABLE 指示的地址就是程序存储区地址，它也可以用常数表示的地址取代。

（3）端口地址（PA）寻址，主要指令有：

```
PORTR   PA,   Smem
PORTW   Smem,   PA
```

PA 寻址时，指令中的一个操作数用符号或者常数来指明其端口地址。例如，要将数据存储区 200h 单元的内容复制到端口地址为 FIFO 的 I/O 口，可以表示为：

```
PORTW   200h,   FIFO
```

其中，FIFO 指示端口地址，同样 FIFO 可以用常数表示的端口地址取代。

（4）*（lk）寻址，适用于所有支持单访问数据存储区（Smem）操作数寻址的指令。它用符号或者常数来指明数据空间地址。例如，要将累加器 A 的低 16 位放入数据存储区地址为 BUFFER 的单元中，可以表示为：

```
STL   A,   *（BUFFER）
```

（lk）寻址允许所有 Smem 寻址的指令无须改变 DP 或初始化 AR 就能够访问数据空间的任何单元。当使用这种寻址方式时，指令的长度扩展 1 个字。例如，一个单字长指令扩展成双字长指令或者 1 个双字长指令扩展成三字长指令。注意：使用（lk）绝对寻址的指令不能和单重复指令 RPT、PRTZ 一起使用。

3．累加器寻址

累加器寻址是将累加器的内容作为一个地址，把程序存储区当做数据存储区来访问。下面两条指令就是采用累加器的寻址方式：

```
READA   Smem
WRITA   Smem
```

READA 是将累加器 A 所指定的程序存储区单元内容传送到 Smem 指向的数据存储区单元中。WRITA 是将 Smem 指向的数据存储区单元内容传送到累加器 A 所指定的程序存储区单元中。

需要注意的是，有些 C54x 芯片的程序存储区单元由累加器的低 16 位确定，但是对于另一些 DSP 芯片，如 C548、C549、C5410、C5416 有 23 根地址线，C5402 有 20 根地址线，C5420

有 18 根地址线，这些芯片的累加器寻址时，其程序存储区单元由累加器的低 23 位确定。

4．直接寻址

直接寻址的指令中，其中的一个操作数指示数据存储区地址（dma）的低 7 位。这 7 位 dma 作为偏移地址和基地址结合，形成 16 位的数据存储区地址。基地址存放在数据页指针寄存器 DP 或者堆栈指针寄存器 SP 中。采用直接寻址无须改变 DP 或 SP 的值就可以任意访问当前基地址中连续的 128 个单元。这种方式的显著特点是指令代码只有 1 个字。图 7-2 给出了使用直接寻址的指令代码格式。

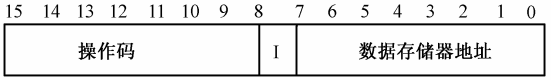


图 7-2　直接寻址的指令代码格式

其中，15~8 位为指令的操作码，第 7 位 I 确定了寻址方式。若 I=0，表示指令使用直接寻址方式。6~0 位包含了指令的数据存储区的偏移地址。

直接寻址的语法是用 1 个符号或 1 个常数来确定偏移值。例如，要将地址为 SAMPLE 的存储区单元内容加到累加器 B 中，可以表示为：

```
ADD   SAMPLE,  B
```

则地址 SAMPLE 的低 7 位存放在指令代码中，高 9 位由 DP 或 SP 提供。具体使用 DP 还是 SP 根据状态寄存器 ST1 中的编译模式位 CPL 来确定。CPL=0，dma 域与 9 位 DP 域结合，形成 16 位的数据存储区地址；CPL=1，dma 域（正偏移）加上 SP 形成 16 位数据存储区地址。

在以 DP 为基地址的直接寻址中，DP 位于数据地址高 9 位，dmad 位于数据地址低 7 位。图 7-3 给出了数据地址的形成过程。

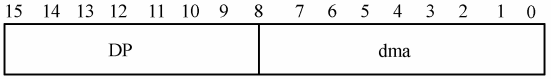


图 7-3　以 DP 为基地址的直接寻址

由于 DP 的范围为 0~511（2^9-1），因此以 DP 为基地址的直接寻址将数据存储区分为 512 页。由于 dma 的范围为 0~127（2^7-1），所以每页有 128 个可以访问的单元。也就是说，DP 指向了 512 页中的一页，而 dma 指向该页 128 个单元中的一个。因此，访问第 1 页的第 0 单元与访问第 2 页的第 0 单元的区别仅仅在于 DP 值改变了。在采用 DP 为基地址的直接寻址时，必须注意首先要对数据页指针进行初始化。

在以 SP 为基地址的直接寻址中，7 位 dma 作为一个正偏移与 SP 相加得到有效的 16 位数据存储区地址。图 7-4 给出了实际地址的形成过程。



图 7-4　以 SP 为基地址的直接寻址

SP 可以指向数据存储区的任意一个地址，dma 指向当前页中一个明确的单元，从而允许访问数据存储区任意基地址中连续的 128 个单元。

5．间接寻址

在间接寻址方式中，数据空间任意单元都可以通过一个辅助寄存器中的内容所代表的 16 位地址进行访问。C54x 有 8 个 16 位的辅助寄存器 AR0-AR7。当指令以间接方式寻址时，辅助寄存器和地址可以有选择地进行增量、减量、偏移或者变址的修改，还可以提供循环和位反转寻址。其中，AR0 除了可以像其他辅助寄存器一样使用外，还能够用于指数寻址和位反转寻址中。两个辅助寄存器算术单元 ARAU0 和 ARAU1 可以根据辅助寄存器的内容进行操作，完成无符号 16 位的算术运算。

间接寻址方式相当灵活，不但可以用一条指令从数据存储区读或写一个 16 位的数据操作数（单操作数寻址），还能在一条指令中访问两个数据存储区单元（双操作数寻址），包括从两个不同的数据存储区单元读数据，读并写两个连续的存储区单元，或者读一个存储区单元同时写另一个存储区单元。

（1）单操作数间接寻址

这种方式可以通过在指令中修改辅助寄存器来改变寻址单元，具体的修改方式有：地址加 1 或减 1、加 16 位偏移量、用 AR0 值作为偏移量等。这些地址修改可以在地址访问之前或者之后进行。加上不修改地址的情况，一共可以形成 16 种寻址方式。表 7-9 列出了单操作数间接寻址的句法以及每一种句法的功能。

表 7-9　单操作数间接寻址类型

句　法	功　能	说　明
*ARx	addr=ARx	ARx 包含数据存储区地址
*ARx-	addr=ARx ARx＝ARx －1	访问完成后，ARx 中的地址递减‡
*ARx+	addr=ARx ARx＝ARx +1	访问完成后，ARx 中的地址递增‡
*+ ARx	addr=ARx+1 ARx＝ARx +1	访问之前，ARx 中的地址递增‡§#
*ARx-0B	addr=ARx ARx＝B（ARx-AR0）	访问完成后，ARx 减去 AR0 的值并进行位反转
*ARx-0	addr= ARx ARx＝ARx-AR0	访问完成后，ARx 减去 AR0 的值
*ARx+0	addr=ARx ARx＝ARx+AR0	访问完成后，ARx 加上 AR0 的值
*ARx+0B	addr=ARx ARx＝B（ARx+AR0）	访问完成后，ARx 加上 AR0 的值并进行位反转
*ARx-%	addr= ARx ARx＝circ（ARx-1）	访问完成后，ARx 中的地址按循环寻址方式递减‡
*ARx-0%	addr= ARx ARx＝circ（ARx-AR0）	访问完成后，ARx 减去 AR0 的值并进行循环寻址
*ARx+%	addr= ARx ARx＝circ（ARx+1）	访问完成后，ARx 中的地址按循环寻址方式递增‡
*ARx+0%	addr=ARx ARx＝circ（ARx+AR0）	访问完成后，ARx 加上 AR0 的值并进行循环寻址
*ARx（lk）	addr=ARx+lk ARx＝ARx	ARx 与 16 位长偏移 lk 之和作为数据存储区地址，ARx 不变

（续表）

句　法	功　能	说　明
*+ARx（lk）	addr=ARx+lk ARx＝ARx+lk	访问前，有符号的 16 位长偏移 lk 加到 ARx 中，然后新的 ARx 作为数据存储区地址§
*+ARx（lk）%	addr=circ（ARx+lk） ARx＝circ（ARx+lk）	访问前，有符号的 16 位长偏移 lk 加到 ARx 中，再循环寻址后存入 Arx。然后新的 ARx 作为数据存储区地址§
*（lk）	addr=lk	16 位无符号偏移量 lk 作为数据存储区的绝对地址§

注：‡——访问 16 位字时，递增/递减值为 1；访问 32 位字时，递增/递减值为 2。

§——不允许用在存储区映射寄存器寻址方式中。

#——只用于写操作。

表 7-9 中涉及循环寻址和位反转寻址两种特殊的寻址过程，下面作一介绍。

- 循环寻址

在卷积、自相关和 FIR 滤波器等许多算法中，都需要在存储区中设置循环缓冲区。循环缓冲区是一个滑动窗口，包含着最近的数据。如果有新的数据到来，它将覆盖最早的数据。实现循环缓冲区的关键是循环寻址。循环寻址用%表示，其辅助寄存器使用规则与其他寻址方式相同。

循环缓冲区的参数主要包括：长度寄存器（BK）、有效基地址（EFB）和尾地址（EOB）。其中，BK 定义了循环缓冲区的大小 R。要求缓冲区地址始于最低 N 位为零的地址，且 R 值满足 $2^N > R$，R 值必须要放入 BK。例如，一个长度为 31 个字的循环缓冲区必须开始于最低 5 位为零的地址（即 XXXX XXXX XXX0 0000b），且赋值 BK=31。又如，一个长度为 32 个字的循环缓冲区必须开始于最低 6 位为零的地址（即 XXXX XXXX XX00 0000b），BK=32。

循环缓冲区的有效基地址（EFB）定义了缓冲区的起始地址，也就是辅助寄存器（ARx）低 N 位设为 0 后的值。循环缓冲区的尾地址（EOB）定义了缓冲区的底部地址，它通过用 BK 的低 N 位代替 ARx 的低 N 位得到。

循环缓冲区的指示 index 就是当前 ARx 的低 N 位，步长 step 就是一次加到辅助寄存器或从辅助寄存器中减去的值。循环寻址的算法为：

```
If   0≤ index + step< BK;
index = index + step;
Else if index + step≥BK;
index = index + step – BK;
Else if index + step <0;
index = index + step + BK;
```

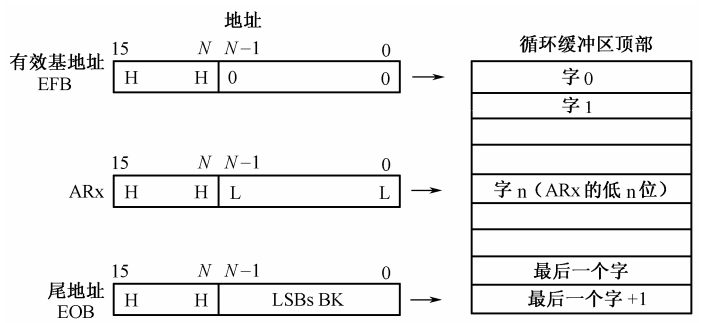


图 7-5　循环缓冲区示意图

例如，对于指令：

```
LD   *+AR1（8）%, A
STL  A, *+AR1（8）%
```

如果循环缓冲区长度 BK＝10，N=4，AR1=100h，由 AR1 的低 4 位得到 index＝0，循环寻址*+AR1（8）%的步长 step＝8，循环寻址过程见图 7-6。

执行第 1 条指令时：index = index + step＝8，寻址 108h 单元。

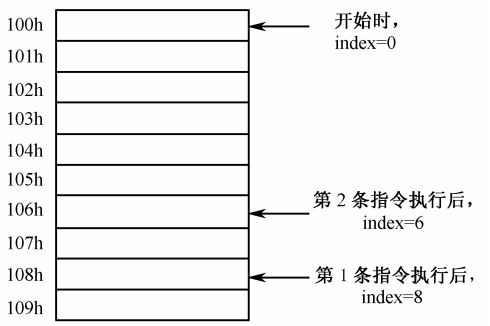


图 7-6　循环寻址示意图

执行第 2 条指令时：index = index + step＝8＋8＝16>BK，故 index = index + step－BK＝8＋8－10＝6，寻址 106h 单元。

使用循环寻址时，必须遵循以下三个原则：

① 循环缓冲区的长度 R 小于 2^N，且地址从一个低 N 位为 0 的地址开始。

② 步长小于或等于循环缓冲区的长度。

③ 所使用的辅助寄存器必须指向缓冲区单元。

循环寻址典型地采用加 1 或减 1（*ARx+%或*ARx-%），或者加减辅助寄存器 AR0 存放的数值（*ARx+0%或*ARx-0%）来实现。循环地址也可由一个立即数来进行修改（*+ARx（lk）%），此时指令代码将为 2 至 3 个字长，在这种情况下，循环不能采用单重复（RPT 和 RPTZ）指令。

- 位反转寻址

位反转寻址主要用于 FFT 算法中，这种寻址方式可以显著提高程序的执行速度和存储区的利用效率。使用时，AR0 存放的整数值为 FFT 点数的一半，另一个辅助寄存器 ARx 指向存放数据的单元。位反转寻址将 AR0 加到辅助寄存器中，地址以位反转方式产生。也就是说，两者相加时，进位是从左向右反向传播的，而不是通常加法中的从右向左。例如，1010 与 1100 的位反转相加结果为 0001：

$$\begin{array}{r} 1\,0\,1\,0 \\ +\ 1\,1\,0\,0 \\ \hline 0\,0\,0\,1 \end{array}$$

假设辅助寄存器为 8 位，AR2 值为 0110 0000b，AR0 的值为 0000 1000b，下面例子给出了位反转寻址中 AR2 值修改的顺序和修改后 AR2 的值。

例：

```
*AR2+0B  ;AR2 = 0110 0000          （第 0 值）
*AR2+0B  ;AR2 = 0110 1000          （第 1 值）
```

```
*AR2+0B  ;AR2 = 0110 0100        （第 2 值）
*AR2+0B  ;AR2 = 0110 1100        （第 3 值）
*AR2+0B  ;AR2 = 0110 0010        （第 4 值）
*AR2+0B  ;AR2 = 0110 1010        （第 5 值）
*AR2+0B  ;AR2 = 0110 0110        （第 6 值）
*AR2+0B  ;AR2 = 0110 1110        （第 7 值）
```

表 7-10 给出了 AR0 为 0000 1000b 时，位模式和位反转模式与 AR2 低 4 位的关系。

表 7-10　位反转寻址

原顺序	位模式	位反转模式	位反转后的顺序
0	0000	0000	0
1	0001	1000	8
2	0010	0100	4
3	0011	1100	12
4	0100	0010	2
5	0101	1010	10
6	0110	0110	6
7	0111	1110	14
8	1000	0001	1
9	1001	1001	9
10	1010	0101	5
11	1011	1101	13
12	1100	0011	3
13	1101	1011	11
14	1110	0111	7
15	1111	1111	15

（2）双操作数间接寻址

双数据存储区操作数寻址用于完成两个读操作或者一个读并行一个写的操作。采用这种方式的指令代码都为 1 个字长并且只能以间接寻址方式工作。由于指令代码中只有两位来表示辅助寄存器，因此只有 4 个辅助寄存器可以被选择：AR2~AR5。两个 ARAU 和 4 个辅助寄存器一起可以在单周期内访问两个操作数。

表 7-11 列出了双数据存储区操作数间接寻址的句法和功能。

表 7-11　双数据存储区操作数间接寻址类型

句 法	功 能	说 明
*ARx	addr=ARx	ARx 为数据存储区地址
*ARx-	addr=ARx ARx＝ARx−1	访问完成后，ARx 中的地址减 1
*ARx+	addr=ARx ARx＝ARx+1	访问完成后，ARx 中的地址加 1
*ARx+0%	addr=ARx ARx＝circ（ARx+AR0）	访问完成后，ARx 加上 AR0 的值并进行循环寻址

6．存储区映射寄存器寻址

这种寻址方式用来修改存储区映射寄存器内容，但是不会影响当前的 DP 或 SP 值。由于不

需要对 DP 或 SP 进行操作，因此这种寻址方式往寄存器写数据所占用的开销是最小的。存储区映射寄存器寻址既可以在直接寻址中使用，又可以在间接寻址中使用。具体的地址产生方法为：

- 在直接寻址方式下，数据存储区地址的低 7 位来自指令字，不管当前 DP 或 SP 值为多少，数据地址的高 9 位都被强制清零；
- 在间接寻址方式下，数据存储区地址的低 7 位来自当前辅助寄存器的低 7 位，数据地址的高 9 位置零，当操作完后，辅助寄存器的高 9 位被强制清零。

例如，如果 AR1 指向一个存储区映射寄存器，AR1 值为 FF25h，那么 AR1 最低 7 位为 25h，所指示的数据存储区地址为 0025h。由于定时器周期寄存器 PRD 的地址为 0025h，这样，AR1 就指向了定时器周期寄存器。执行完毕后，存放在 AR1 中的值改变为 0025h。

此外，这种寻址方式除了能够修改寄存器以外，数据第 0 页中的便笺式 RAM 的任意单元都可以采用存储区映射寄存器寻址来进行修改。

汇编指令中，只有 8 条指令能使用存储区映射寄存器的寻址方式：

```
LDM   MMR,   dst
MVDM   dmad,   MMR
MVMD   MMR,   dmad
MVMM   MMRx,   MMRy
POPM   MMR
PSHM   MMR
STLM   src,   MMR
STM   #lk,   MMR
```

需要注意的是，在存储区映射寄存器寻址方式下，不能采用下列间接寻址的句法：*ARx（lk），*+ARx（lk），*+ARx（lk）％和*（lk）。

7．堆栈寻址

系统堆栈在调用子程序或者中断时能够自动保存程序计数器 PC 值，它也可以用来保存程序当前的环境或要传递的数据。堆栈存放数据是从高端地址向低端地址进行的。DSP 用 1 个 16 位的堆栈指针 SP 来对堆栈寻址，SP 总是指向堆栈中最后存入的数据单元。下面 4 条语句采用了堆栈寻址的方式来访问堆栈：

- PSHD——将一个数据存储区的值压入堆栈；
- PSHM——将一个存储区映射寄存器的值压入堆栈；
- POPD——将一个数据存储区的值弹出堆栈；
- POPM——将一个存储区映射寄存器的值弹出堆栈。

图 7-7 给出了将一个数据 X2 压入堆栈（PSHD X2）的操作过程，从图中可以清楚看出堆栈和指针 SP 在操作前后的变化情况。对 C54x DSP 来说，数据压入堆栈前要对 SP 进行减量运算，而在数据弹出堆栈操作之后，要对堆栈进行增量运算。

在中断和调用子程序过程中，堆栈用来存放和恢复 PC 值。当一个中断产生或者调用一个子程序时，返回地址会自动压入堆栈顶部。调用中断和子程序的指令有：CALA[D]、CALL[D]、CC[D]、INTR 和 TRAP。中断返回或者调用子程序返回时，返回地址从堆栈中弹出，存放到 PC 中。返回指令有：RET[D]、RETE[D]、RETEF[D]和 RC[D]。此外，FRAME 指令也能影响堆栈，它将一个短立即数偏移量与 SP 指针相加，从而修改 SP 的值。在直接寻

址中，也可以用堆栈指针 SP 作为基地址来寻址。

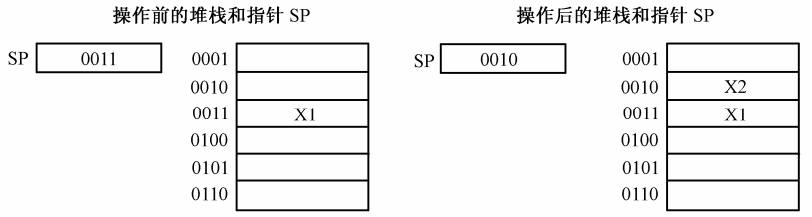


图 7-7 数据压入堆栈操作示意图

7.4.2 程序寻址

在程序存储区中存放着程序代码、系数表和立即操作数等信息，访问这些信息时需要使用程序寻址方式。程序寻址主要包括程序存储区地址如何产生以及程序地址如何装入程序计数器 PC。

1．程序存储区地址产生

C54x 通过程序地址总线 PAB 来对程序存储区进行访问的，早期的 C54x 芯片只能访问 64K 字的程序空间。随着 DSP 程序地址总线的扩展，DSP 能够访问的片内外的程序空间逐渐变大。表 7-12 给出了几种常见的 C54x 系列芯片的访问程序空间情况。

表 7-12 DSP 程序总线以及访问程序空间情况

DSP 芯片	扩展的地址总线	最大访问空间
C548、C549、C5409，C5410，C5416	7	128 个 64 K 字的程序页
C5402	4	16 个 64 K 字的程序页
C5420	2	4 个 64 K 字的程序页

程序地址产生逻辑（PAGEN）产生程序地址并将其放入程序地址总线 PAB。PAGEN 包括 5 个寄存器：程序计数器 PC、重复计数器 RC、块重复计数器 BRC、块重复起始寄存器 RSA 和块重复结束地址寄存器 REA。此外，C548、C5410 和 C5416 等 DSP 还有一个程序计数器扩展寄存器 XPC，用来对扩展程序存储区进行寻址。C54x DSP 取指令时将 PC 值放入 PAB，然后读取相应的程序存储单元中的指令。在读取这个程序存储区单元时，PC 值递增，为下一次取指令作准备。当程序执行转移、调用、返回、中断或循环指令时，程序地址会出现不连续的情况，此时相应的目标地址装入 PC。通过 PAB 寻址得到的指令代码接着被装入指令寄存器 IR 中。

为了改善某些指令的性能，程序地址产生逻辑 PAGEN 也可以从程序存储区中读取操作数，例如读取系数表中某个数据或者将数据写入数据表中某个单元，或者将数据在程序空间和数据空间之间传送。此外还有一些指令可以利用程序总线来取出另一个乘数，比如 FIRS、MACD 和 MACP 等指令。

2．程序计数器

程序计数器 PC 是一个包含内部或外部程序存储区地址的 16 位寄存器，PC 控制着程序的运行过程。在通常情况下，程序指令按前后顺序逐条执行时，PC 值依次递增；但是当程序指令出现转移、子程序调用、子程序返回、条件操作、单指令重复、指令块重复、硬件复位

或者中断操作时，PC 值会出现不连续的情况。此时 PC 值的变化情况见表 7-13。

表 7-13　各种情况下的 PC 变化

指令操作	装入 PC 的地址
复位	FF80h → PC
顺序执行	PC+1 → PC
转移	转移指令中的 16 位立即数值放入 PC
以累加器内容为地址转移	累加器 A 或 B 的低 16 位放入 PC
块重复循环	当 BRAF=1 时，若 PC+1=（REA）+1，则 RSA 的内容装入 PC
子程序调用	PC+2 的值压入堆栈顶部，然后调用指令中的 16 位立即数地址放入 PC。返回指令将堆栈顶部数据弹出并放到 PC，从而使程序返回到调用的下一条指令
以累加器内容为地址进行子程序调用	PC+1 的值压入堆栈顶部，然后累加器 A 或 B 的低 16 位放入 PC。返回指令将堆栈顶部数据弹出并放到 PC，从而使程序返回到调用的下一条指令
中断	PC 值压入堆栈，相应的中断向量地址放入 PC。中断返回时将堆栈顶部数据弹出并放到 PC，从而使程序返回到中断前的指令处

3．扩展程序计数器

扩展程序计数器（XPC）是一个 7 位寄存器。在 C548、C549、C5410、C5416 和 C5420 等 DSP 芯片中，使用 XPC 可以对程序存储区扩展空间进行寻址。在程序计数器 PC 被装入时，XPC 值也随情况发生变化。表 7-14 给出了各种情况下 XPC 的变化。

表 7-14　各种情况下 XPC 的变化

指令操作	装入 PC 的地址
复位	FF80h → PC，XPC=0
顺序执行	PC+1 → PC，XPC 不变
长转移	转移指令中的立即数的 15~0 位放入 PC，23~16 位放入 XPC
以累加器内容为地址长转移	累加器 A 或 B 的 15~0 位放入 PC，23~16 位放入 XPC
子程序长调用	PC+2 的值压入堆栈顶部，然后 XPC 压入堆栈顶部，最后调用指令中的立即数的 15~0 位放入 PC，23~16 位放入 XPC
以累加器内容为进行子程序长调用	PC+1 的值压入堆栈顶部，然后 XPC 压入堆栈顶部，最后累加器 A 或 B 的 15~0 位放入 PC，23~16 位放入 XPC
长返回	首先将堆栈顶部数据弹出并放到 XPC 中，然后再将堆栈顶部值弹出并放到 PC 中，从而使程序返回到调用时的指令处

4．延时转移

DSP 芯片在执行指令时采用了流水线结构，执行一条指令被分成了取指、译码和执行 3 个部分。也就是说，当一条指令运行到执行部分时，后续的一条双字长指令或者两条单字长指令就已经进入了流水线。因此，当一条延时指令运行到执行部分时，后续的一条双字长指令或者两条单字长指令也已经进入了流水线。C54x DSP 中的汇编指令在执行转移操作时都分为延时转移和不延时转移两种方式，在转移指令上加后缀 D 表示延时转移。延时转移指令能够使紧跟转移指令之后的一条双字长指令或两条单字长指令执行完毕后，再跳转到指令指定的地址执行。而对于不延时转移指令，它直接跳转到指令指定的地址执行，此时紧跟在转移指令之后的一条双字长指令或两条单字长指令将不被执行，直接在流水线上被冲洗掉。由于冲洗流水线需要附加周期，降低了程序运行效率，因此正确使用延时转移和不延时转移指令

可以提高程序的运行速度。

7.5 小　　结

本章以助记符指令集为重点，详细介绍了 TMS320C5000 的指令。对汇编语言源程序的格式和汇编指令进行了总体介绍，并详细说明了汇编语法和指令使用方法。最后，重点讲解了 TMS320C5000 的寻址方式。掌握 TMS320C5000 的汇编语言并深刻理解 DSP 的寻址方式对于软件编程、提高编写程序的运行效率有着重要意义。

7.6 习题与思考题

（1）汇编语句格式包含哪几部分？编写汇编语句需要注意哪些问题？

（2）C54x 的指令集包含了哪几种基本类型的操作？

（3）C54x 提供了哪些基本的数据寻址方式？

（4）以 DP 和 SP 为基地址的直接寻址方式，其实际地址是如何生成的？

（5）循环寻址时必须遵循哪几个原则？

（6）采用*AR2+0B 寻址，若 AR0 为 0000 1000b，试写出位模式和位反转模式与 AR2 低 4 位的关系。

第 8 章　基于 C 和汇编语言混合编程的 DSP 芯片开发

8.1　引　　言

第 5 章讨论了基于 C 语言开发 DSP 芯片的方法。用 C 语言开发 DSP 程序不仅使 DSP 开发的速度大大加快，而且 DSP 程序的可读性和可移植性也显著提高，程序修改也非常方便。采用 C 编译器的优化功能可以提高 C 代码的效率，在某些情况下，C 优化的效率甚至接近手工优化的效率。用 C 语言开发 DSP 程序，在 DSP 芯片资源不是十分紧张时是非常合适的。

虽然 C 编译器的优化功能可以提高 C 代码的效率，但是在某些情况下，C 代码的效率还是无法与手工编写的汇编代码的效率相比，如 FFT 程序。这是因为，即使是优化的 C 编译器，也无法在所有的情况下都能够最佳地利用 DSP 芯片所提供的各种资源，如 TMS320C54x 所提供的循环寻址、用于 FFT 的比特反转寻址等。用 C 语言编写的中断程序虽然可读性很好，但由于在进入中断程序后，有时不管程序中是否用到，中断程序都将寄存器进行保护，从而降低了中断程序的效率。如果中断程序频繁被调用，那么即使是一条指令也是至关重要的。此外，用 C 语言实现 DSP 芯片内的某些功能模块控制也不如汇编程序方便，有些甚至无法用 C 语言实现。

因此，在很多情况下，DSP 应用程序往往需要用 C 语言和汇编语言的混合编程方法来实现，以达到最佳利用 DSP 芯片软硬件资源的目的。

8.2　基于 C 和汇编语言的混合编程方法

用 C 语言和汇编语言的混合编程方法主要有以下三种：

（1）独立编写 C 程序和汇编程序，分开编译或汇编，形成各自的目标代码模块，然后用链接器将 C 模块和汇编模块链接起来。例如，FFT 程序一般采用汇编语言编写，对 FFT 程序用汇编器进行汇编，形成目标代码模块，与 C 模块链接就可以在 C 程序中调用 FFT 程序。

（2）直接在 C 语言程序的相应位置嵌入汇编语句。

（3）对 C 程序进行编译，生成相应的汇编程序，然后对汇编程序进行手工优化和修改。

下面我们分别介绍这三种方法。

8.2.1　独立的 C 和汇编模块接口

这是一种常用的 C 和汇编语言接口方法。采用这种方法时需注意的是在编写汇编语言和 C 语言时必须遵循有关的调用规则和寄存器规则。如果遵循了这些规则，那么 C 和汇编函数之间的接口是非常方便的。C 程序既可以调用汇编程序，也可以访问汇编程序中定义的变量。同样，汇编程序也可以调用 C 函数或访问 C 程序中定义的变量。

8.2.1.1 TMS320C3x/C4x

混合编写 TMS320C3x/C4x 的 C 和汇编程序时，必须注意如下几点：

（1）不论是用 C 语言或是用汇编语言编写的函数，必须遵循函数调用规则和寄存器规则。

（2）必须保护用于函数修改的几个特定的寄存器，其中，DP 只在应用小模式时保护，而 R8 是对 C4x 而言的。这些特定的寄存器为：R4, R5, R6, R7, AR4, AR5, AR6, AR7, FP, DP, SP, R8。在这些寄存器中，除了 R6 和 R7 用于存放浮点数之外，其余寄存器都用来存放整数。此外，若 SP 正常使用，则不一定保护，也就是说，只要压入堆栈的数在函数返回时全部弹出，则汇编函数就可自由地使用堆栈。其他寄存器可自由使用而不必保护。

（3）在中断程序中，必须保护所有用到的寄存器。其中，R0~R7（对 C4x 为 R0~R11）必须保护全部 40 位，因为这些寄存器包含的可能是整数或浮点数。

（4）访问堆栈中的变量或特定寄存器中的变量取决于使用的变量传递模式。

（5）从汇编调用 C 函数时，需按照 C 调用时保存变量的方法保护变量，即按相反的次序将变量压入堆栈，最右边说明的变量首先保护，最左边的变量最后压入，这样就使得最左边的变量位于堆栈的顶部。采用寄存器－参数模式时，将参数装入特定的寄存器，按照有关规则将剩下的参数压入堆栈。

（6）必须根据 C 函数的说明类型返回正确的函数值。整数、指针和浮点值在寄存器 R0 中返回。

（7）C 汇编模块只能将 .cinit 块用于变量的初始化，而不能用于其他目的。boot.asm 中的 C 启动程序认为 .cinit 块完全是由初始化表所构成的。如果在 .cinit 块中存入其他信息会导致不可预测的后果。

（8）C 编译器将 C 程序中定义的所有标识符在其前面加上一个下划线“_”。因此，在汇编模块中，必须要在 C 程序中访问的变量或标识符前加一下划线。例如，在 C 程序中的变量 varc，在汇编中调用时为 _varc。同样，要定义一个能在 C 程序中访问的变量 vara，汇编程序中的定义应为 _vara。当然，如果一个变量仅在汇编模块中使用，则变量名前不加下划线也可以任意使用，不会与 C 标识符发生冲突。

（9）若要定义能在 C 程序中调用或访问的汇编函数或变量，必须在汇编程序中用 .global 命令说明为外部变量。同样，在汇编中要调用 C 函数或访问 C 变量，在汇编程序中也必须将它们说明为外部变量。

8.2.1.2 TMS320C2x/C5x

与 TMS320C3x/C4x 一样，在编写独立的 TMS320C2x/C5x 汇编模块的过程中，最重要的是必须遵守 TMS320C2x/C5x 定点 C 编译器所定义的函数调用规则和寄存器使用规则。遵循了这两个规则就可以保证所编写的汇编模块不破坏 C 的运行环境。C 模块和汇编模块可以相互访问各自定义的函数或变量。在编写独立的汇编程序时，必须注意以下几点：

（1）不论是用 C 编写的函数还是用汇编编写的函数，都必须遵循寄存器使用规则。

（2）必须保护函数要用到的几个特定寄存器。在 TMS320C2x/C5x 的 C 编译器中，这些特定的寄存器包括：AR0（FP）, AR1（SP）, AR6 和 AR7。其中，如果 SP 正常使用，则不必明确加以保护。换句话说，只要汇编函数在函数返回时弹出压入的对象，实际上就已经保护了 SP，因而 AR1 就可以自由地使用。其他寄存器是可以自由使用的。函数返回时 ARP 必须为 1。

（3）中断程序必须保护所有用到的寄存器。

（4）从汇编程序调用 C 函数时，以逆序方式将参数压入堆栈，调用之后，再将参数弹出。

（5）调用 C 函数时，C 函数只保护几个特定的寄存器，而对于其他寄存器 C 函数是自由使用的。

（6）长整型和浮点数在存储器中存放的顺序是低位字在低地址，高位字在高地址。

（7）如果函数有返回值，则返回值存放在累加器 ACC 中。

（8）汇编模块不能改变由 C 程序产生的 .cinit 块，如果改变其内容则会引起不可预测的后果。

（9）编译器在所有的标识符（函数名、变量名等）前要加一下划杠“_”。因此，编写汇编语言程序时，必须在 C 程序可以访问的所有对象前加“_”。例如，在 C 程序中定义了变量 x，如果在汇编程序中要使用，即标记为“_x”。如果仅在汇编中使用，则只要不加下划杠，即使与 C 程序中定义的对象名相同，也不会造成冲突。

（10）任何在汇编程序中定义的对象或函数，如果需要在 C 程序中访问或调用，则必须用汇编指令 .global 定义。同样，如果在 C 程序中定义的对象或函数，需要在汇编程序中访问或调用，在汇编程序中也必须用 .global 指令定义。

8.2.1.3　TMS320C54x

编写可以被 C/C++函数调用，或可以调用 C/C++函数或变量的独立 TMS320C54x 汇编模块，必须注意以下几点：

（1）不论是用 C 编写的函数还是用汇编编写的函数，都必须遵循寄存器使用规则。

（2）必须保护函数要用到的几个特定寄存器。在 TMS320C54x 的 C 编译器中，这些特定的寄存器包括： AR1、AR6、AR7 和 SP。其中，若 SP 是被正常使用，则不必明确加以保护，其他寄存器则可以自由使用。

（3）中断程序必须保护所有用到的寄存器。

（4）从汇编程序调用 C 函数时，第一个参数必须放在累加器 A 中。其他参数以逆序方式压入堆栈。调用之后，将参数弹出。

（5）调用 C 函数时，注意 C 函数只保护了几个特定的寄存器，而其他寄存器 C 函数是自由使用的。

（6）从 C 程序调用汇编函数时，第 4 点规则同样适用，除非 C 语言的参数声明中使用省略号以表明其参数数目是可变的。此时，若除了省略号外只有一个参数被声明，则这个参数不再通过累加器 A，而是通过堆栈来传递。如：

```
int vararg （ int first , …… ）;
```

其中的参数 first 将不再通过累加器 A，而是通过堆栈来传递。

（7）长整型和浮点数在存储器中存放的顺序是高位字在低地址，低位字在高地址。

（8）如果函数有返回值，则返回值存放在累加器 ACC 中。

（9）汇编模块不能改变由 C 产生的 .cinit 块，如果改变其内容则会引起不可预测的后果。

（10）编写汇编语言程序时，必须在 C 程序可以访问的所有对象前加“_”。例如，在 C 程序中定义了变量 x，如果在汇编程序中要使用，即为_x。如果某个对象仅在汇编中使用，则只要不加下划杠，即使与 C 程序中定义的对象名相同，也不会造成冲突。

（11）任何在汇编中定义的对象或函数，如果需要在 C 中访问或调用，则必须用汇编指令 .global 定义。同样，如果在 C 中定义的对象或函数，需要在汇编中访问或调用时，在汇编中也必须用 .global 指令定义。

（12）一些 C54x 指令一次能够读写或保存 32-bit 的数据，如指令 DLD, DADD 等。为保证 32-bit 数的读写正确，在函数调用时，必须保证 SP 指针为偶数。

（13）C 语言编写的程序在编译汇编后，将 CPL 置为 1，因此其中调用的汇编函数中若需访问直接地址对象，唯一方法就是使用间接的绝对地址模式。如：

```
LD   *（global_var）, A          ;工作于 CPL＝1
LD   global_var, A               ;工作于 CPL＝0
```

（14）若汇编函数中将 CPL 设置为 0，则在返回 C 语言编写的程序前，必须将 CPL 置为 1。

（15）C 语言编写的程序无法通过 IMR 寄存器对中断进行允许或屏蔽。此时，实现中断管理的方法就是采用如 8.2.4 节中的在 C 语言中嵌入汇编指令的方法来进行。

（16）利用 C 语言编写中断服务程序时，前面需加上中断关键字 interrupt，如 interrupt void isr（）。

（17）为了将一个中断程序与一个中断相关联，必须在相应的中断矢量处放置一个跳转。可以使用链接器和汇编器，在.sect 块中创建一个简单的跳转指令表来达成这个目标。



8.2.2 从 C 程序中访问汇编程序变量

从 C 程序中访问在汇编程序中定义的变量或常数需根据变量或常数定义的方式采取不同的方法。总的来说，可以分为三种不同的情形：变量在 .bss 块中定义；变量不在 .bss 块中定义；常数。

对于访问在 .bss 块中定义的变量，可用如下方法实现：

（1）采用 .bss 命令定义变量；

（2）用.global 命令定义为外部变量；

（3）在变量名前加一下划线“_”；

（4）在 C 程序中将变量说明为外部变量。

采用上述方法后，在 C 程序中就可以访问这个变量。例 8.1 示出了在 C 程序中访问 .bss 定义的汇编变量。

【例 8.1】在 C 程序中访问 .bss 定义的汇编变量

汇编程序：

```
        .bss        _var,1      ;定义变量
        .global     _var        ;说明为外部变量
```

C 程序：

```
        extern  int  var;       /*外部变量*/
        var =1;                 /*访问变量*/
```

对于访问不在 .bss 块中定义的变量，其方法比较复杂一些。在汇编中定义的常数表是这种情形的一个常见例子。在这种情况下，必须定义一个指向该变量的指针，然后在 C 程序中间接地访问这个变量。在汇编中定义常数表时，可以为这个表定义一个独立的块，也可以在现有的块中定义。定义完后，说明一个指向该表起始的全局标号。如果定义为一个独立的块，则可以在链接时将它分配至任意可用的存储器空间。在 C 程序中访问该表时，必须另外说明一

个指向该表的指针。需要注意的是，对于 TMS320C3x/C4x，当使用优化 C 编译时，必须使用 -mf 选项以使优化器保留间接存储器访问。在默认的情况下，TMS320C3x/C4x 的 C 优化器认为所有的变量都定义在 .bss 块中，从而有可能将间接存储器访问优化为直接存储器访问。例 8.2 是一个在 C 程序中访问汇编常数表的例子。

【例 8.2】在 C 程序中访问汇编常数表

汇编程序：

```
.global   _sine            ;定义外部变量
.sect     "sine_tab"       ;定义一个独立块
_sine:                     ;常数表起始地址
        .float      0.00
        .float      0.02
        .float      0.02
        .float      0.03
```

C 程序：

```
extern  float sine[];                  /*定义外部变量*/
float   *sine_pointer=sine;            /*定义一个 C 指针*/
f = sine_pointer[2];                   /*访问 sine_pointer*/
```

对于在汇编中用 .set 和 .global 命令定义的全局常数，也可以从 C 程序中访问，不过访问的方法更复杂些。一般对于在 C 或汇编中定义的变量，符号表实际上包含的是变量值的地址，而非变量值本身。然而，对于在汇编中定义的常数，符号表包含的是常数的值。而编译器不能区分符号表中哪些是变量值，哪些是变量的地址。因此，在 C 程序中访问汇编中的常数不能直接使用常数的符号名，而应在常数名之前加一个地址操作符“&”。如在汇编中的常数名为“_x”，则在 C 程序中的值应为“&x”。如例 8.3 中所示。

【例 8.3】在 C 程序中访问汇编常数

汇编程序：

```
_table_size   .set   10000          ;常数定义
.global  _table_size                ;定义为全局
```

C 程序：

```
extern int table_size;
#define   TABLE_SIZE   ((int)  (&table_size))
      ⋮
      ⋮
for（i=0;   i<TABLE_SIZE; ++i）
```



8.2.3　在汇编程序中访问 C 程序变量

在编写独立的汇编程序时，经常需要访问在 C 程序中定义的全局变量或数组。下面两个例子介绍了如何在汇编程序中访问 C 程序定义的变量和数组。

【例 8.4】访问在 C 程序中定义的全局变量（TMS320C3x）

C 程序：

```
int          i;
float        x;
main（）
```

```
{
}
```

汇编程序：

```
.global   _i              ;/*定义 i 为全局变量*/
.global   _x              ;/*定义 x 为全局变量*/
LDI       @_i,R0          /*R0=i*/
STF       R1,@_x          /*x=R1*/
```

【例 8.5】访问在 C 程序中定义的全局数组（TMS320C3x）

C 程序：

```
int       h[20];
float     s[80];
main（）
{
}
```

汇编程序：

```
.global                _h                ;/*定义_h 为全局变量*/
.global                _s                ;/*定义_s 为全局变量*/
.data
h_add     .word        _h                ;/*数组 h 的起始地址*/
s_add     .word        _s                ;/*数组 s 的起始地址*/
.text
LDI            0,R0
STI            R0,@_h+1                  /*h[1]=0*/
LDF            @_s+10,R1                 /*R1=s[10]*/

LDI            @h_add,AR0                 /*AR0=数组 h 的起始地址*/
LDI            @s_add,AR1                 /*AR1=数组 s  的起始地址*/
LDI            0,R0
LDF            0.0,R1
RPTS           19
STI            R0,*AR0++                  /*for（i=0;i<20;i++）  h[i]=0*/
RPTS           79
STF            R1,*AR1++                  /*for（i=0;i<80;i++）  s[i]=0.0*/
```



8.2.4 在 C 程序中直接嵌入汇编语句

在 C 程序中嵌入汇编语句是一种直接的 C 和汇编接口方法。采用这种方法一方面可以在 C 程序中实现用 C 语言无法实现的一些硬件控制功能，如修改中断控制寄存器、中断使能或无效、读取状态寄存器和中断标志寄存器等。另一方面，也可以用这种方法在 C 程序中的关键部分用汇编语句代替 C 语句以优化程序。

采用这种方法的一个缺点是比较容易破坏 C 环境，因为 C 编译器在编译嵌入了汇编语句的 C 程序时，并不检查或分析所嵌入的汇编语句。

嵌入汇编语句的方法比较简单，只须在汇编语句的左右加上一个双引号，用小括弧将汇编语句括住，在括弧前加上 asm 标识符即可，如下所示。

```
asm（“      汇编语句              ”）;
```

如上所述，在 C 程序中直接嵌入汇编语句的一个典型应用是控制 DSP 芯片的一些硬件资源。如 TMS320C54x，在 C 程序中可用下列汇编语句实现一些硬件控制：

```
asm（"    SSBX    INTM    "）；/*关中断*/
asm（"    LD      #0, DP  "）；/*DP=0*/
asm（"    LD      #10, A  "）；/*A=10*/
asm（"    SSBX    OVM     "）；/*打开溢出保护*/
asm（"    SSBX    XF      "）；/*置 XF 为高电平*/
asm（"    RSBX    INTM    "）；/*开中断*/
```

除了硬件控制之外，汇编语句也可以嵌入在 C 程序中实现其他功能。TI 建议不要采用这种方法改变 C 变量的数值，因为这容易改变 C 环境，但是如果程序员对 C 编译器及 C 环境具有充分的理解，并且小心使用，采用这种方法也可以对 C 变量进行自由地操作。与独立编写汇编程序实现混合编程相比，这种方法具有如下优点：

（1）程序的入口和出口由 C 自动管理，不必手工编写汇编程序实现。

（2）程序结构清晰。由于这种方法保留了 C 程序的结构，如变量的定义等，因此程序结构清晰，可读性强。

（3）程序调试方便。由于 C 程序中的变量全部由 C 来定义，因此，采用 C 源码调试器可以方便地观察 C 变量。

下面以 C54x 为平台，举例说明用汇编语句嵌入 C 程序实现变量操作。

【例 8.6】在 C 程序中嵌入汇编语句

```
long   add_sub（int,int*）;
main（）
{
  int i,k;
  int a,b[3];
  long result;

  a=1;
  for （i=0;i<3;i++）        b[i]=i+1;
  result = add_sub（a,b）;
}
long add_sub（int a, int*b）
{
  int   i;
  long sum;

  asm（"   SSBX      SXM          "）;
  asm（"   NOP                    "）;
  asm（"   LD       *SP（6）,B    "）;
  asm（"   STLM     B,AR1         "）;
  asm（"   NOP                    "）;
  asm（"   NOP                    "）;
  asm（"   RPT      #2            "）;
  asm（"   ADD      *AR1+,A       "）;
  asm（"   DST      A,*SP（2）    "）;
```

```
    asm（"    NOP                    "）;

    return（sum）;
}
```

需要特别注意的是，采用这种方法后，对程序进行编译时不能采用优化功能，否则将产生不可预测的结果。



8.2.5 修改 C 编译器的输出

我们可以通过控制 C 编译器使编译器产生具有交叉列表的汇编程序。例如通过下列编译可产生具有交叉列表的汇编程序 function.asm:

```
TMS320C3x:
    cl30   -v30   -s   function.c
TMS320C5x:
    dspcl   -v50 -s   function.c
TMS320C54x:
    cl500   -s   function.c
```

在所生成的 function.asm 中，C 程序中的语句作为说明出现在汇编程序中，每个 C 语句前都加上“;>>>>”的说明符，每个 C 语句的下面是 C 编译器对该 C 语句编译所生成的汇编语句，如例 8.7 中所示。

【例 8.7】交叉列表的汇编程序（TMS320C2x）

```
;>>>>                  main（）
;>>>>                  int i,j;
*****************************************
*          FUNCTION DEF :    _main          *
*****************************************
_main:
    POPD     *+
    SAR      AR0,*+
    SAR      AR1,*
    LARK     AR0,3
    LAR      AR0,*0+,AR2
;>>>>                  i+=j;
    LARK     AR2,1
    MAR      *0+
    LAC      *+
    ADD      *-
    SACL     *
;>>>>                  j=i+123;
    LAC      *+
    ADDK     123
    SACL     *,AR1
EPIO_1:
    SBRK     4
    LAR      AR0,*-
```

```
PSHD    *
RET
.end
```

通过查看交叉列表的汇编程序，可以对某些编译不是很优但却比较关键的汇编语句进行修改。修改汇编语句时，必须严格遵循不破坏 C 环境的原则。

修改完后，对汇编程序进行汇编就可以产生目标文件。

8.3　基于 C 和汇编语言的混合编程举例



8.3.1　TMS320C3x/C4x 混合编程举例

本节以 TMS320C3x 为例，说明浮点 DSP 芯片的混合编程方法。

【例 8.8】C 程序调用汇编函数

C 程序：

```
/*cmain.c*/
extern int asmfunc（）;          /*定义外部汇编函数*/
int     gvar;                    /*定义全局变量*/
main（）
{
        int    i;
        i=asmfunc（i）;          /*调用汇编函数*/
}
```

汇编程序：

```
/*asmsub.asm*/
FP      .set   AR3               ;FP=AR3
        .global _asmfunc         ;定义外部函数
        .global      _gvar       ;定义外部变量
        .text
_asmfunc:
.if       .REGPARM=0             ;标准运行模式
PUSH      FP                     ;保护 FP
LDI       SP,FP                  ;指向栈顶
LDI       *-FP（2）,AR2          ;变量装入到 AR2
.endif
LDP       _gvar                  ;设定 DP 等于 gvar 的页
LDI       @_gvar,R0
ADDI      AR2,R0                  ;R0＝gvar+变量值
.if       .REGPARM=0
          POP        FP          ;恢复 FP
.endif
          RETS
.end
```

对上述程序，首先对 C 主程序 cmain.c 进行编译，形成 cmain.obj:

```
cl30    -v30   -o2   cmain.c
```

然后对 asmsub.asm 汇编子程序进行汇编，形成 asmsub.obj:

```
asm30 -v30 asmsub.asm
```

最后，对两个目标模块进行链接即可形成可执行程序。



8.3.2 TMS320C2x/C5x 混合编程举例

【例 8.9】TMS320C2x 的独立 C 和汇编模块编程

C 主程序：

```
/*cmain.c*/
extern  int  asmfunc（）;          /*定义外部的汇编函数*/
int          gvar;                 /*定义全局变量*/
main（）
{
        int  i;
        i=asmfunc（i）;            /*调用函数*/
}
```

汇编语言函数 _asmfunc:

```
/*asmsub.c*/
.global  _asmfunc
.global  _gvar
.text
_asmfunc:
POPD     *+                ;将返回地址保存至 C 堆栈
SAR      AR0,*+            ;保护 FP
SAR      AR1,*             ;保护 SP
LARK     AR0,1             ;帧的长度
LAR      AR0,*0+,AR2       ;建立 FP 和 SP

LDPK     _gvar             ;指向_gvar
SSXM                       ;置符号扩展
LAC      _gvar             ;装入 gvar 至 ACC
LARK     AR2,-3            ;参数偏移
MAR      *0+               ;指向参数
ADD      *,AR0             ;将参数加至 gvar
SACL     _gvar             ;保存在 gvar 中

LARP     AR1
SBRK     2
LAR      AR0,*-            ;恢复帧指针
PSHD     *                 ;返回地址压入 C2x 堆栈
RET
.end
```

对上述程序，编译汇编和链接的过程如下。

（1）对 C 程序 cmain.c 进行编译形成 cmain.obj:

```
dspcl -v25 -o2 cmain.c
```

（2）对汇编程序 asmsub.asm 进行汇编形成 asmsub.obj:

```
dspa   -v25   asmsub.asm
```

（3）链接形成可执行程序。

8.3.3　TMS320C54x 混合编程举例

【例 8.10】TMS320C54x 的独立 C 和汇编模块编程

C 主程序：

```
/*cmain.c*/
extern   int   asmfunc（）;          /*定义外部的汇编函数*/
int              gvar;               /*定义全局变量*/
main（）
{
        int   i;
        i=asmfunc（i）;              /*调用函数*/
}
```

汇编语言函数 _asmfunc:

```
FP      .set   AR7
        .mmregs
        .global    _asmfunc
        .global    _gvar
        .bss          _gvar,1,0,0
_asmfunc:
         ;* A     assigned to _i                ;累加器 A 分配给变量 i
         FRAME       #-1                        ;（SP）-1→SP: 形成局部帧
         NOP                                    ;避免流水线冲突
         STL         A,*SP（0）                 ; 参数 i 暂存到 SP（0）中
         ADD          *（_gvar）,A              ;将参数与 gvar 相加
         STL         A,*（_gvar）               ;保存至 gvar
         FRAME       #1                         ;（SP）+1→SP:撤销局部帧
         RET                                    ;函数返回
         .end
```

对上述程序，编译汇编和链接的过程如下：

（1）对 C 程序 cmain.c 进行编译形成 cmain.obj:

```
cl500    -o2 cmain.c
```

（2）对汇编程序 asmsub.asm 进行汇编形成 asmsub.obj:

```
asm500   asmsub.asm
```

（3）链接形成可执行程序。

8.4　DSP 代码的优化

DSP 芯片是为了高效地进行数字信号处理而设计的，其中包含两方面的内容：第一，要实现运算时间上的最少；第二，要实现存储空间的最少。为达到这个目的，DSP 芯片中有针对性地设计了一些特殊的功能电路，如硬件乘法器、硬件无开销循环等，以用最少的指令开销和

时间开销实现某些功能。这些功能电路的高效应用往往涉及多方面的考虑，C 编译器的规则有时难以完全覆盖，因此所得到的代码并非是最优的。此时，开发者就需要对代码进行手工优化。

首先，代码的优化从 C 语言程序的编写开始就需进行。其次，在 C 编译时，选择不同的编译选项，也将对 C 程序进行不同程度的优化。最后，针对 C 程序编译得到的汇编代码，也可以进一步进行手工优化，以充分利用 DSP 芯片的各种特殊功能模块。

在第 5 章中曾介绍过，在使用 C 语言编译器对源程序进行编译的时候，编译器可以根据程序员的要求对程序进行适当的优化，例如将循环语句简化、在寄存器中保存变量等等，以充分使用上述的 DSP 硬件电路，提高代码的运行速度，减少代码的占用空间。详细的内容请参阅第 5 章和相关 DSP 芯片的用户手册。但是，这种优化的实现是有限的，尤其在是面对大型而复杂的应用软件时。

本节主要介绍 C 代码的优化和汇编代码的优化。



8.4.1 C 代码的优化

在 C 代码的函数编写中，遵循以下几点，将有助于提高执行代码的效率。

1．尽量避免使用除法运算和取模运算

在 DSP 芯片中，并没有硬件除法电路，因此除法和取模运算，需要调用除法库函数。而 DSP 库函数中的除法都以移位减的方式实现，实现的代码多，占用的运算时间长。

若除法运算中的除数为常数时，可以将除法运算转换成乘法运算，由于 DSP 芯片中硬件乘法器的存在，可以大大减少代码量和运算时间。同理，取模运算可以通过减法加条件跳转来实现，当模数为 2 的幂时，也可以通过逻辑运算来实现，这两种实现方法都能有效减少取模运算的代码量和运算时间。

2．慎用嵌套循环

采用编译优化选项，可以将嵌套循环的最内部的循环用单指令循环或块循环等无开销循环方式取代，如 C54x 芯片中的 RPT 指令和 RPTB 指令。但由于块循环不能嵌套使用，因此其外层循环仍然只能用减法和条件跳转的方式实现，当循环次数较多时，其所占用指令周期也随之增大。因此，在内部循环次数不多、运算简单的情况下，以连续排列运算代码的方式取代循环体，有可能反而不增加甚至减少代码量，并减少执行时间。

3．尽量减少变量的设置

在代码运行期间，C 代码中的全局变量将始终占据数据区的存储空间，而局部变量也在定义范围内占据数据区。大部分 DSP 芯片内部的存储区有限，适合分配为数据区的片内 RAM，特别是片内 DARAM 区更为有限。因此，变量设置过多，将大大增加对 DSP 芯片内部 RAM 存储区域的需求，减少 DSP 芯片的使用灵活性。

此外，由于 C 代码中的全局变量和静态变量一般在编译后由绝对寻址方式访问，与其他寻址方式相比，这将占用大量的指令空间，因此应尽量减少全局变量和静态变量的设置。

4．尽量避免在循环中调用函数

由于循环中的函数调用，需要保护循环操作的环境，因此编译器无法生成包含函数调用的硬件无开销循环，而这将降低循环代码段的编译效率。

5．遵循 5%－80%规则

经过长期的统计观察，人们发现，对于大部分的程序，其执行时间的 80%是由其 5%的代码占用的。因此，在 C 代码的编写中，可以利用 DSP 编译环境 CCS 中的 profile 功能对 C 代码进行分析，针对上述 5%的 C 代码进行分析优化，其效果将优于对其他代码段的优化。

6．尽量采用高效的库文件或内联函数进行运算

C 语言并没有和芯片结构相关联，直接使用 C 语言编程，无法利用 DSP 特定的硬件结构优势。为了方便用户使用 C 语言开发 DSP 程序，TI 等公司提供了运行支持库（run_time support library）文件 rts.lib，其源码保存在 rts.src 文件中。这个文件包含了 ISO 定义的标准运行支持函数、编译工具函数、浮点代数运算函数和 C 语言 I/O 函数等。

TI 公司还为 C54x、C55x 和 C6000 等平台提供了 DSPLIB 函数库，其中包含了数十个 C 可调用的汇编通用信号处理函数，这些函数都针对各自平台进行了优化，因此相比于直接使用标准 ANSI C 语言编写的代码，其执行时间被大大地缩短了。为了使用这个库文件，需要在 C 文件中包含 dsplib.h 头文件。

此外，DSP 芯片公司还利用汇编语言为不同的 DSP 芯片实现了常用代数运算函数，以供 C 语言调用。这些函数称为内联函数（inline function）。编译器能够识别内联操作函数。这些内联函数可以像函数一样使用，并可以产生利用 C/C++语言无法表达的汇编指令语句。这些内联操作函数在符号最前面有一个下划线，以和普通函数有所区别，但在使用上和普通函数没有差别，可以正常地调用。利用这些内联函数完成代数运算，可以充分利用 DSP 的汇编指令，实现高效的程序代码。在 CCS 的编译选项中，选择－o3 优化，将自动嵌入内联函数。

需要注意的是，内联函数与库函数在最终得到的代码上是以不同形式被使用的。库函数都以函数调用的方式被使用，而内联函数则是在代码中的相应位置直接插入所使用的内联函数代码。从上面的描述可以看出，内联函数由于是直接插入代码，因此无需函数调用的相应操作，能节省执行时间，但是当多次被使用时，其代码将被多次插入，从而增加了算法的代码量。因此，内联函数与库函数的选择，应当从执行时间和代码空间两方面综合进行考虑。在编译器的选项中可以使用－oi 选项来控制内嵌函数代码的大小。

表 8-1 是 C54x 芯片中常用的 C 语言内联函数列表。

表 8-1　C54x 芯片中常用的 C 语言内联函数

内联函数	汇编指令	功能说明
short _abs（short src）;	ABS	产生一个 16 位绝对值
long _labs（long src）;	ABS	产生一个 32 位绝对值
short _abss（short src）;	ABS	产生一个溢出保护的 16 位绝对值。当 OVM 设置时，_abss（0x8000） = 0x7FFF
long _labss（long src）;	ABS	产生一个溢出保护的 32 位绝对值，当 OVM 设置时，_labss（0x8000000） = 0x7FFFFFFF
short _addc（short src1, short src2）;	ADDC	将 src1,src2 和进位一起累加，和为 16 位数据
long _laddc（long src1,short src2）;	ADDC	将 src1,src2 和进位一起累加，和为 32 位数据
short _norm（short src）;	EXP	产生一个可用于归一化 src 的左移移位数
short _lnorm（long src）;	EXP	产生一个可用于归一化 src 的左移移位数
short _rnd（long src）;	RND 或 ADD	利用 src+2^{15} 对 src 进行舍入。当 OVM 设置时，产生 16 位溢出保护结果
short _sadd（short src1, short src2）;	ADD	累加两个 16 位整数，当 OVM 设置时产生 16 位溢出保护结果

（续表）

内联函数	汇编指令	功能说明
long _lsadd（long src1, long src2）;	ADD	累加两个 32 位整数，当 OVM 设置时产生 32 位溢出保护结果
long _smac（long src, short op1, short op2）;	MAC	op1 和 op2 相乘后左移 1 位并与 src 相累加，结果为 32 位溢出保护数据（OVM 和 FRCT 设置）
short _smacr（long src, short op1, short op2）;	MACAR	op1 和 op2 相乘后左移 1 位并与 src 相累加，然后累加 2^{15} 对结果进行舍入（OVM 和 FRCT 设置）
long _smas（long src, short op1, short op2）;	MAS	op1 和 op2 相乘后左移 1 位，并从 src 中减去，结果为 32 位溢出保护数据（OVM 和 FRCT 设置）
short _smasr（long src, short op1, short op2）;	MASAR	op1 和 op2 相乘后左移 1 位，并从 src 中减去，然后累加 2^{15} 对结果进行舍入（OVM 和 FRCT 设置）
short _smpy（short src1, short src2）;	MPYA	src1 和 src2 相乘并左移 1 位，结果为 16 位溢出保护数据（OVM 和 FRCT 设置）
long _lsmpy（short src1, short src2）;	MPY	src1 和 src2 相乘并左移 1 位，结果为 32 位溢出保护数据（OVM 和 FRCT 设置）
short _smpyr（short src1, short src2）;	MPYR	src1 和 src2 相乘并左移 1 位，然后累加 2^{15} 对结果进行舍入（OVM 和 FRCT 设置）
short _sneg（short src）;	NEG	对 16 位数据取反，溢出保护 _sneg（0xffff8000） = 0x00007FFF
long _lsneg（long src）;	NEG	对 32 位数据取反，溢出保护 _lsneg（0x80000000） => 0x7FFFFFFF
short _sshl（short src1, short src2）;	SFTA	将 src1 左移 src2 位得到 16 位的结果，如果 src2 小于等于 8，结果进行溢出保护（OVM 设置）
long _lsshl（long src1, short src2）;	SFTA	将 src1 左移 src2 位得到 32 位的结果，如果 src2 小于等于 8，结果进行溢出保护（OVM 设置）
short _ssub（short src1, short src2）;	SUB	从 src1 中减去 src2，OVM 设置有效，得到 16 位溢出保护结果
long _lssub（long src1, long src2）;	DSUB	从 src1 中减去 src2，OVM 设置有效，得到 32 位溢出保护结果
short _subc（long src1, short src2）;	SUBB	从 src1 中减去 src2 和符号位的逻辑逆，结果为 16 数据
long _lsubc（long src1, short src2）;	SUBB	从 src1 中减去 src2 和符号位的逻辑逆，结果为 32 数据

在 CCS 的编译路径包含的头文件 intrindefs.h 中，给出了 ETSI 支持的常见函数预定义，其中的函数大多数都采用了内联函数形式实现，如下所示。

```
#define L_add（a,b） （_lsadd（（a）,（b）））
#define L_sub（a,b） （_lssub（（a）,（b）））
#define L_negate（a） （_lsneg（a））
#define L_deposit_h（a） （（long）a<<16）
#define L_deposit_l（a） （a）
#define L_abs（a） （_labss（（a）））
#define L_mult（a,b） （_lsmpy（（a）,（b）））
#define L_mac（a,b,c） （_smac（（a）,（b）,（c）））
#define L_macNs（a,b,c） （L_add_c（（a）,L_mult（（b）,（c））））
#define L_msu（a,b,c） （_smas（（a）,（b）,（c）））
#define L_msuNs（a,b,c） （L_sub_c（（a）,L_mult（（b）,（c））））
#define L_shl（a,b） （（b）<0？L_crshft（（a）,（-b））：（b）<9？_lsshl（（a）,（b））: L_clshft（（a）,（b））)
#define L_shr（a,b） （L_crshft（（a）,（b））)
#define L_shr_r（a,b） （L_crshft_r（（a）,（b））)
#define abs_s（a） （_abss（（a））)
```

```
#define add（a,b）　（_sadd（（a）,（b）））
#define sub（a,b）　（_ssub（（a）,（b）））
#define extract_h（a）　（（unsigned）（（a）>>16））
#define extract_l（a）　（（int）a）
#define round（a）　（_rnd（a））
#define mac_r（a,b,c）　（_smacr（（a）,（b）,（c）））
#define msu_r（a,b,c）　（_smasr（（a）,（b）,（c）））
#define mult（a,b）　（_smpy（（a）,（b）））
#define mult_r（a,b）　（_smpyr（（a）,（b）））
#define norm_s（a）　（_norm（a））
#define norm_l（a）　（_lnorm（a））
#define negate（a）　（_sneg（a））
#define shl（a,b）　（clshft（（a）,（b）））
#define shr（a,b）　（crshft（（a）,（b）））
#define shr_r（a,b）　（crshft_r（（a）,（b）））
#define div_s（a,b）　（divs（a,b））
```

8.4.2　汇编代码的优化

8.4.2.1　TMS320C2x 汇编优化技巧

1．充分利用片内RAM

TMS320C2x 大部分指令的执行时间是 1 个指令周期，但是在实际应用中，根据所采用的程序存储器和数据存储器是片内还是片外，实际执行的时间是不相同的。按照指令在内部存储器和外部存储器以及数据在内部存储器和外部存储器，可以分为 4 种情况，即：（1）指令在内部存储器、数据在内部存储器（PI/DI）；（2）指令在内部存储器、数据在外部存储器（PI/DE）；（3）指令在外部存储器、数据在内部存储器（PE/DI）；（4）指令在外部存储器、数据在外部存储器（PE/DE）。设 p 表示外部程序存储器所需的等待状态数，d 表示外部数据存储器所需的等待状态数，则执行 LAC 指令在 4 种情况下所需的周期数如表 8-2 所示。

表 8-2　LAC 指令所需的周期数

PI/DI	PI/DE	PE/DI	PE/DE
1	2+d	1+p	2+d+p

显然，即使外部的存储器是高速存储器，不需要等待状态，在 PI/DE 和 PE/DE 两种情况下，执行 LAC 所需的时间也是 2 个指令周期，而不是 1 个指令周期。而 PI/DI 和 PE/DI 两种情况是 1 个指令周期。因此，当程序在外部存储器中运行时，应尽量使用片内 RAM 存储器存储数据。虽然 1 条指令只相差 1 个周期，但当指令在多级嵌套循环内部时就相当可观了。

2．利用单指令重复指令

TMS320C2x 提供了重复指令 RPT 和 RPTK，这两条指令可以使下一条指令重复执行指定的次数而不增加额外的执行时间。TMS320C2x 的重复计数器为 8 位，因此最大值为 255，即最多可以使下一条指令执行 256 次。RPT 和 RPTK 与 TBLR/TBLW, BLKD/BLKP, IN/OUT 及 MAC, MACD 等指令结合使用可以提高程序运行的效率。例如下面的一段程序将存放在外部数

据存储器地址 400h 开始的 256 个数据搬移至片内的 B1 块。

```
LARP    AR1
LRLK    AR1，300h
RPTK    255
BLKD    400h,*+
```

3．利用高效指令和特殊寻址方式

TMS320C2x 的指令集中提供了一些高效的指令。如乘累加并移位的 MACD 指令、平方并累加或减指令 SQRA 和 SQRS 以及 LACT, LTA, LTD, SPAC, APAC 等。除了常用的寻址方式外，还提供了用于 FFT 的比特反转寻址方式，灵活使用这些指令和寻址方式可以提高程序的效率。

8.4.2.2 TMS320C5x 汇编优化技巧

TMS320C5x 的指令集是 TMS320C2x 指令集的超集，它在 TMS320C2x 的基础上作了改进并增加了一些功能更强的指令，充分利用这些增强特性可大大提高 TMS320C5x 的程序效率。

1．利用片内存储器的双映射特性

TMS320C5x 具有 1 056 字的片内双访问存储器，另一种存储器是单存访问储器，TMS320C53 和 C51 分别具有 3K 字和 1K 字的单访问存储器，而 C50 具有 8 K 字。这种存储器块的一个特点是同时可以映射至程序和数据空间，这种双映射的特点对于实现诸如自适应 FIR 滤波器（如横向滤波器）非常有用。乘/加循环要求 FIR 滤波器的系数在程序空间中，但同时相同的系数表需在数据空间中被更新。将这个系数表存放于片内的单访问存储器中并利用其双映射特点可使横向滤波器实现更高的效率。

2．利用无开销循环

在 TMS320C2x 中，只提供了用于单指令重复的 RPT/RPTK 指令，要实现多于 1 条指令的循环必须采用寄存器不为零跳转的 BANZ 指令，这样每次循环实际上增加了 3 个指令周期。TMS320C5x 不仅提供了单指令重复指令 RPT，而且提供了块指令重复指令 RPTB，因此，TMS320C5x 可以实现块指令的无开销循环。例如：

```
    LACL    #99
    SAMM    BRCR            ;循环 100 次
    RPTB    LOOP-1
    LAC     *,16,AR1
    MPYA    *+,AR2
    SACH    *0-
LOOP:
```

3．嵌套循环的高效编程

在许多 DSP 应用程序中需要用到嵌套循环。如求 10 阶自相关需要用到二级嵌套循环，内循环计算每个自相关系数，外循环进行计数。如果内循环可以用 RPT 实现，则外循环可以用 RPTB 指令实现。如果内循环不能用一个 RPT 实现循环，也可以用延迟跳转循环来实现（如 AR 不为零，延迟跳转 BANZD）。最多可以有 8 个这样的 BANZD 循环嵌套，每个循环用一个辅助寄存器作循环计数器。

4．使用CRGT和CRLT搜索最大/最小值

TMS320C5x 提供了用于高效搜索最大和最小值的特殊指令 CRGT 和 CRLT，这两个指令比较累加器 ACC 和累加器缓冲器 ACCB 之间的大小，比较之后将最大或最小值存放于 ACCB 中。例如下面一段程序用于在 300H 至 309H 存储单元中搜索最大值并得到其地址。

```
MAXADR   .set   60h
MAXVAL   .set   61h
         .text
         LDP      #0              ;DP=0
         LAR      AR0,#0300h
         SETC     SXM
         LACC     #08000h         ;装入一个最小值
                                  ;（如果找最小值则装入 7FFFH）
         SACB                     ;送至 ACCB
         SPLK     #9,BRCR         ;重复 10 次
         RPTB     endb-1
         LACC     *               ;取数至 ACC
         CRGT                     ;比较（如果找最小值用 CRLT 指令）
         SACL     MAXVAL          ;保存最大值
         XC       #1,C
         SAR      AR0,MAXADR      ;保存地址
         MAR      *+
endb     RET
```

5．利用循环寻址方式

TMS320C5x 提供了循环寻址方式，这种寻址方式在实现 FIR 滤波器时非常有用。一般实现自适应 FIR 滤波器采用具有数据移动操作的乘/加指令 MACD，在 TMS320 C5x 中，可以用循环寻址方式代替数据移动操作来更新滤波器的抽头系数。如果滤波器的抽头在片内的单访问存储器中，则这种方式的运行速度更快。

6．利用延迟跳转和条件运行

TMS320C5x 具有 4 级深度的流水线。C5x 的 4 级深度流水线使芯片的并行性能更强。然而，流水线越深，清空流水线的周期越长。正常跳转时，由于需要清空流水线，因此需要 4 个周期才能实现。但是，对 C5x 而言，所有需要清空流水线的指令都支持延迟执行功能，使运行开销降低至 2 个周期。此外，如果是仅跳转 1 个或 2 个指令这种特殊情况，可以使用速度更快的条件运行指令 XC，此时仅需 1 个周期。

7．并行逻辑单元

TMS320C5x 的比特操作单元与算术逻辑单元是独立运行的。这可使 C5x 在没有 ACC 或 ACCB 参与的情况下对片内或片外存储器单元（包括存储器映射的寄存器）进行逻辑操作。例如：

```
SMMR     DRR, #DRR_DATA
LMMR     DXR, #DXR_DATA
OPL      #RXDATA, SP_FLG
APL      #TXDATA, SP_FLG
```

8．增强的移位功能

TMS320C2x 具有较强的移位功能，如取操作数至 ACC 时带 0~16 位左移、存 ACC 时带 0~7 位左移、根据 TREG1 寄存器进行左移的取数指令 LACT、ACC 左移一位 ROL 和 ACC 右移一位 ROR 等。TMS320C5x 在此基础上增加了 ACC 右移 1~16 位的指令 BSAR 和根据 TREG1 寄存器值进行 16 位右移指令 SATH 及 0~15 位右移指令 SATL。例如：

```
BASR    7           ;ACC=ACC>>7
LMMR    TREG1,1BH
SATH                ;ACC=ACC>>16，因为 TREG1.4=1
SATL                ;ACC=ACC>>11
```

此外，C5x 还具有对 ACC、ACCB 和进位位组成的 65 位数据进行移位的指令。

9．存储器映射寄存器

TMS320C5x 提供了累加器与寄存器之间的直接通路，这是将寄存器映射至数据存储器而实现的。同时，TMS320C5x 也支持存储器和寄存器之间的直接数据交换。例如：

```
SPLK    #16, INDX
SAMM    PRD
```

10．系数表的动态寻址

在用乘/加指令实现 FIR 滤波器时，TMS320C5x 从数据存储器取输入样值，而从程序空间取滤波器系数。由于同时从存储器中取两个操作数，使乘/加操作得以在一个周期内完成。在 C2x 中，系数表的地址只能用直接寻址方式指定，这可满足大部分的应用。但是如果系数表的地址事先不能确定，而需在程序运行过程中确定时，C2x 实现就比较困难。对这种情况，C5x 提供了一个称为块移动地址寄存器（BMAR）的专用寄存器，使乘/加操作可以用间接方式寻址，BMAR 指向程序存储器中的系数表地址。例如：

```
LACL    table_add
SAMM    BMAR        ; 更新系数表的地址
LAC     ONE,14
ZPR                 ; PREG=0
RPT     num_a-1     ; 重复
MADD    *－         ; 乘/加
APAC                ; 加最后一个乘积
SACH    result,1    ; 保存结果
```

8.4.2.3 TMS320C54x 汇编优化技巧

在编写 TMS320C54x 程序时，首先必须正确进行编译模式的选择。在 ST1 状态寄存器中，有 1 位编译器模式控制位 CPL。CPL 用于指示在相对直接寻址中采用哪种指针。当 CPL=0 时，使用页指针 DP；当 CPL=1 时，采用堆栈指针 SP。

1．流水线冲突

在 TMS320C54x 中，采用了深度为 6 级的流水线操作，因此，流水线冲突是不可避免的。一般情况下，当发生流水线冲突时，由 DSP 自动插入延迟解决冲突问题。但有些情况下 TMS320C54x DSP 无法自动解决冲突问题，需要程序员通过调整程序语句的次序或在程序中插入一定数量的 NOP 来解决。这一点对于刚开始进行 C54x 编程的用户来说十分不便，但却

非常重要。如果你在调试程序时不能得到正确结果，而又找不出程序错误时，就应该想到是否发生了流水线冲突，解决方法就是在合适的位置插入一至几个 NOP 指令。有关何时需要程序员插入 NOP 指令，读者可参阅 C54x 介绍资料的第 1 卷（VOL.1），这里就不一一罗列了。下面我们举几个例子予以说明。

```
（1）  STLM      A，BRC
       NOP
       RPTB      endloop-1
          .
          .
       endloop:
（2）  RSBX      SXM
       NOP
       LD        *AR5+，A
（3）  STLM      A，ST1
       NOP
       SUB       *AR2-，A
（4）  SSBX      CPL              ; CPL=1
       NOP
       NOP
       NOP
       LD        27H，A
（5）  ORM       #8H，PMST        ; 置 DROM=1
       NOP
       NOP
       NOP
       LD        *AR3，A          ; 从片内 ROM 读数据
```

2．利用片内存储空间

TMS320C54x 系列芯片内部集成了不同容量的 RAM 和 ROM，如 TMS320VC5402 的片内 RAM 为 16 K，TMS320VC5409 的片内 RAM 为 32 K，而 TMS320VC5416 的片内 RAM 可达 128K。编写程序时，针对这些不同的存储容量，可以作不同的芯片选择和存储空间安排，以便更有效地使用片内的存储器，节省程序运行时从片外存储器读入程序或数据的时间。

3．利用扩展的程序寻址空间

TMS320C54x 中的很多芯片的程序寻址空间最大可达 8M，其程序地址寻址总线宽度为 23 位。为实现这个功能，在指令集中扩充了 6 个特别指令：FB, FBACC, FCALA, FCALL, FRET, FRETE，同时还增加了一个特别的存储器映射寄存器——程序计数器扩展寄存器（XPC）。在这两个芯片中，程序空间被划分为 128 页，每页为 64 K 空间。利用这些特别的指令和寄存器，可以方便迅速地在 8 M 程序空间中寻址，从而可以执行更大规模的程序。需要注意的是，在应用了这些扩充指令后，READA 和 WRITA 的指令地址也扩展到了 23 位。

4．利用TMS320C54x中的两个累加器

TMS320C54x 中集成了两个累加器，它们可以在一个指令周期内同时操作。在 TMS320C54x 系列芯片的指令集中提供了一些指令，可以充分利用这个特性，有效地提高编程

效率。如均方差指令（SQDST）在一个指令周期内，用 B 累加器作为最终寄存器进行矢量差的平方和计算，同时在 A 累加器中进行下一对数值的差值计算。其他还有如 MAX, MIN, FIRS, LMS 等指令，充分利用这些指令，可以减少运算所需的指令周期，提高编程效率。

5．利用ALU的分裂操作模式和CSSU单元

TMS320C54x 的 ALU 可以在一个周期内进行 32 位加法，或进行双 16 位加法，利用这一点，可以加速 Viterbi 蝶形计算。利用 DADST 和 DSADT 指令，可以同时在一个累加器中进行两次运算。CSSU 单元可以在一个指令周期内对某个累加器的高位和低位进行比较，并将比较结果反映在 TRN 寄存器中。利用以上的加速性能，可以在 4 个指令周期内完成一次 Viterbi 蝶形运算。例如：

```
VITRBF   .MACRO
DADST    *AR5,A          ; A = OLD_M（2*J）+T//OLD_（2*J+1）–T
DSADT    *AR5+,B         ; B = OLD_M（2*J）–T//OLD_（2*J+1）+T
CMPS     A,*AR4+         ; NEW_M（J） = MAX（A_HIGH,A_LOW）
                         ; TRN<<1, TRN（0） = TC
CMPS     B,*AR3+         ; NEW_M（J+8） = MAX（B_HIGH,B_LOW）
                         ; TRN<<1, TRN（0） = TC
.ENDM
VITRBR   .MACRO
DSADT    *AR5,A          ; A = OLD_M（2*J）–T//OLD_（2*J+1）+T
DADST    *AR5+,B         ; B = OLD_M（2*J）+T//OLD_（2*J+1）–T
CMPS     A,*AR4+         ; NEW_M（J） = MAX（A_HIGH,A_LOW）
                         ; TRN<<1, TRN（0） = TC
CMPS     B,*AR3+         ; NEW_M（J+8） = MAX（B_HIGH,B_LOW）
                         ; TRN<<1, TRN（0） = TC
.ENDM
```

6．利用自动溢出保护功能

当 SMAU=1 时，在使用 MAC 或 MAS 指令时，首先对乘法结果进行溢出保护，即 8000H ×8000H 得到结果 7FFF FFFFH，然后再进行并行的加或减操作。利用这个特性，可以在 1 个指令周期内完成对乘法结果进行溢出保护后再进行加、减的运算。

7．利用条件存储指令

TMS320C54x 指令集中有 4 个条件存储指令：CMPS, SACCD, SRCCD, STRCD，其中 CMPS 指令在上面已提到。利用这些指令可以减少条件判断指令的开销，提高编程效率。

8．利用循环优化

TMS320C54x 提供了一些零开销的循环指令，如 RPT、RPTB（D）和 RPTZ 等，这类指令的循环次数或者直接给出，或者存放在块重复计数器 BRC 中，由标号确定循环体的结束。为进一步提高效率，在采用这类循环指令的循环体中，需要充分使用辅助寄存器、累加器等。数组尽可能用寄存器间接寻址，中间变量尽可能存放于累加器，以避免中间变量存放的开销。程序员可以根据用户变量的类型和使用频率分配寄存器。循环体内的变量优先级高，使用不重叠的变量可分配给同一个寄存器。具有特定要求的变量可以固定分配给某个寄存器。

9．使用循环寻址

在卷积、自相关和 FIR 滤波器等许多算法中，都需要在存储区中设置循环缓冲区。在这些算法中，循环缓冲区是一个滑动的窗，它包含最近的数据。如果有新的数据到来，它将覆盖最旧的数据。实现循环缓冲器的关键是实现循环循环寻址。循环循环寻址用%表示，其辅助寄存器使用规则与其他寻址方式相同。

10．使用并行指令

在程序中，如果使用双操作数寻址方式，采用并行指令可以提高运行效率。双数据存储器操作数寻址用在完成 2 个读或 1 个读并行 1 个存储的指令中。这些指令只有 1 个字长且只能以间接寻址的方式工作。例如：

```
ST      src, Ymem
|| LD    Xmem, dst
```

其中，用 Xmem 和 Ymem 来代表两个数据存储器操作数。Xmem 表示读操作数；Ymem 在并行读指令时表示读操作数，在一个读并行一个写的指令中 Ymem 表示写操作数。如果源操作数和目的操作数指向同一个单元，那么在并行存储指令中（如 ST||LD），读在写之前执行。如果一个双操作数指令（如 ADD）指向了同一辅助寄存器，而这两个操作数的寻址方式不同，那么就用 Xmem 确定的间接寻址方式来寻址。

11．充分利用乘累加指令

TMS320C54x 提供了一系列的乘累加指令，利用这些指令不仅可以在一个周期内完成一次乘法运算并与累加器相加，而且可以实现一些附带功能。充分利用这些指令，可以减少运算所需的指令周期。在一些需计算乘积和的算法（如滤波）中，乘累加指令得到了广泛的应用。

常用的乘累加指令有：

MAC〔R〕：完成乘累加运算，带后缀 R 时，结果还要进行舍入处理。

MACSU Xmem，Ymem：一个不带符号的数据存储器值 Xmem 与一个带符号的数据存储器值 Ymem 相乘，乘积与累加器的值相加，结果放入累加器。

此外还有 MACA[R], MACD, MACP 等指令完成相应的乘累加运算。与此相对应，MAS[R], MASA[R]等指令完成乘累减功能，在实际中得到了大量使用。

8.4.2.4　TMS320C55x 汇编优化技巧

1．有效使用双MAC硬件

C55x 芯片中包含两个硬件的 MAC 单元，使用好这两个单元，将会有效提高执行效率。在使用中必须注意以下几点：

（1）双 MAC/MAS 操作只能使用三个操作数，也就是说 MAC/MAS 操作要共用其中的一个操作数；

（2）共用操作数必须使用 XCDP（系数指针）寻址，并保存在内部存储器中；

（3）共用操作数应与其他两个操作数存放在不同的存储块中。

2．充分使用并行执行的特性

C55x 结构允许编程者并行放置两个操作或指令，以减少执行时间。并行可分为两类：单指令内建并行和用户自定义并行。其中，单指令内建并行是自动的，用户自定义并行是可选的。

充分利用好 C55x 芯片的并行执行特性，将会大大提高指令的执行效率。具体的并行规则请参考相应的芯片手册。

3．实现高效循环

C55x 芯片中有四种常用的指令循环实现方式：

（1）单循环：repeat（CSR/k8/k16），[CSR+=TA/k4]；

（2）局部块循环：localrepeat{}；

（3）块循环：blockrepeat{}；

（4）当辅助寄存器非零时跳转：if（ARn_mod != 0） goto #loop_start。

前三种方法无须循环开销，后一种方法需要 5 个周期的循环开销。总体上，最有效的循环机制是前两种。

4．最小化流水线和IBQ延迟

C55x 芯片中包含两条分离的受保护的流水线：一条是取指流水线，一条是执行流水线。在流水线中，多条指令将同时执行，在受保护情况下，如果同时执行的指令在数据访问发生冲突时，会自动对其中的某条指令插入额外的周期以防止发生错误。为避免或减少这样的保护带来的执行时间的增加，可以采取适当的措施来预防这样的流水线保护，如指令重排、使用 CPU 寄存器、使用局部循环、重新安排变量和数组在内存中的位置等。

8.5 小 结

本章首先分别介绍了 C 代码和汇编代码混合编程的方法、C 代码中直接嵌入汇编语句的方法和修改编译得到的汇编的方法，随后介绍了编写高效 C 代码的注意事项，最后分别介绍了基于 C2x、C5x、C54x 和 C55x 平台的汇编指令的优化方法。

8.6 习题与思考题

（1）为什么通常需要采用 C 语言和汇编语言的混合编程方法？

（2）C 语言和汇编语言的混合编程方法主要有哪几种？各有什么特点？

（3）整形变量 var 和整形数组 table[10]已经在汇编程序中定义为全局变量和全局数组，试说明如何在 C 程序中访问 var 变量和 table 数组并编写相应的 TMS320C54x 程序。

（4）在汇编程序中如何访问 C 程序中定义的全局变量和全局数组？试编写 TMS320C54x 程序进行实际验证。

（5）修改编译器输出的汇编程序是常用的提高程序效率的方法，试说明采用这种方法的具体步骤以及需要注意的地方。

（6）编写一个基于 C54x 的利用查表法实现正弦函数的 C 语言程序，不用任何编译优化选项进行编译，对得到的汇编程序进行手工优化，并将结果与利用不同编译优化选项得到的汇编程序进行代码量和执行时间的比较。

第 9 章　DSP 芯片集成外设的开发

9.1　引　　言

DSP 芯片除了提供高效的 CPU、丰富的片内 RAM 资源外，还为大量的应用场合设置了多种类型的外设接口（如图 2-14 DSP 结构框图所示）。开发者可以利用这些集成外设接口方便与各种芯片相连，共同实现特定功能的电路系统。从另一个方面来说，片内 CPU 的运行结果只有通过接口和其他器件完成输入输出才有实际意义。因此，一个完善的系统必须要能够对 DSP 的集成外设进行正确合理的使用。可以说，DSP 芯片集成外设的开发是 DSP 芯片系统设计的一个重点。

在 DSP 芯片的 CPU 和外设以及不同的存储空间进行信息（数据、状态信号和控制命令）传送时，为了保证传送的可靠和提高工作效率，有几种不同的传送方式：查询（polling）传送方式、中断（interrupt）传送方式、直接存储器访问（Direct Memory Access，DMA）传送方式。

（1）查询（polling）传送方式

CPU 与外设工作通常是异步的，CPU 的操作和外设信号很难保证同时准备好。因此，可以利用程序控制的传送方式：在传送前，CPU 查询外设的状态，外设准备好了才传送；若未准备好，则 CPU 进行等待。不论数据的传送是输入还是输出，它们的处理方式都相同。

为了实现查询传送方式，芯片除了提供数据传送的端口外，还要提供能够反映端口状态的信号。

（2）中断（interrupt）传送方式

利用查询传送方式，CPU 要不断地查询外设，当外设没有准备好时，CPU 需要等待。如果外设工作速度较慢，CPU 的执行效率就大大降低。为了提高 CPU 的效率，可以采用中断传送方式。当 CPU 需要进行数据传送时，不用去查询外设的状态，当相应的外设准备好数据传送后时会自动发出相应的中断请求，CPU 接收到中断请求后响应这个中断：暂停正在执行的程序、并转去执行中断服务，中断服务处理完后返回到原来的程序继续执行。

中断输入输出方式在单片机、通用 CPU 之类的处理器芯片中都得到了普遍采用。

（3）DMA 传送方式

无论是查询传送方式和中断传送方式，都需要 CPU 参与数据的处理。而如果在存储空间中有大量的数据需要搬移，或者存在高速外设与 CPU 进行成组数据交换的情况，CPU 参与数据传送将占用大量的 CPU 资源，使得 CPU 将长时间处于数据传送状态，而无法处理其他的信号计算任务。因此，如果能够利用硬件，在外设和存储空间之间直接进行数据交换，而不经过 CPU，即采用直接存储空间访问（DMA）传送方式，将可以解放 CPU，使得 CPU 在执行程序的同时，实现大量的数据交换，这一点对于数字信号处理来说十分重要。因此目前的 DSP 等处理器芯片都具备 DMA 功能。

DSP 芯片的 CPU 对外设的控制都是利用寄存器操作实现的。每个外设有对应的控制寄存器或控制比特位。通过设置相应的比特，实现对外设不同功能的选择。同时，为了便于对外设

状态的控制和检测，还为集成外设设置了状态指示寄存器或相应的指示比特位。对于外设需要及时得到 DSP 芯片 CPU 的处理时，特别是外设出现故障需要及时修复时，必须要具有中断控制器，为 CPU 处理有关中断事务（提出中断请求，进行中断优先级排队等），因此很多的集成外设都有对应的中断控制器，尤其是数据通信的外设。

第 2 章中对 DSP 芯片上的集成外设进行了简单介绍。接口的硬件电路只提供了接口工作的平台，要使得 DSP 的含集成外设能够正确地工作，还要对其进行相应的配置。通常需要分初始化和功能设计两个部分实现集成外设的控制程序。本章将重点围绕 C5000 系列 DSP 的硬件开发，对各种集成外设的配置进行详细的介绍，并讨论这些外设的开发方法。

9.2 中　　断

中断是处理器的 CPU 和外部设备进行输入/输出的有效方法。使用中断，可以大大提高 CPU 的效率，实现资源的共享和并行处理，同时也可以在芯片运行过程中对突然发生的故障做出及时发现和处理。

在 DSP 的应用中，常常需要系统能够对信号作出实时响应并能够对信号的变化作出实时性的控制。实时性的工作需要，要求 DSP 芯片能够对各种变量作出快速响应并及时处理，对此，只有靠中断技术才能够有效实现。因此绝大多数的 DSP 集成外设都提供了中断控制机制。

了解 DSP 芯片的中断运行机制，合理运用 DSP 芯片的中断功能，是 DSP 应用系统高效工作的重要保证。本节以 C5000 系列为例，介绍 DSP 芯片的中断控制和开发使用。

9.2.1 概述

中断处理是大多数 DSP 应用系统中不可缺少的一个重要环节。利用中断可以实现 CPU 和外设的同步操作、信号的实时响应和运行故障的快速处理等功能。

9.2.1.1 中断源

引起中断的原因，以及发出中断申请的来源，称为中断源。中断源可以是外部设备的输入输出请求，也可以是芯片内部的外设接口发出的请求，或是 CPU 异常事件等内部原因。

根据中断源的性质不同，DSP 芯片的中断可以分为硬中断和软中断两类，其中，硬中断可以分为外部设备中断、片内外设中断以及其他一些故障中断等多种类型。外部中断来自 DSP 芯片的外部条件，如 I/O 设备或者其他芯片，这些设备以完全随机的方式中断当前程序转入另一个处理程序。片内外设中断包括片内集成设备产生的定时中断、DMA 中断、McBSP 传输中断等。故障中断包括总线错误中断等，主要用于开发者调试过程中问题定位使用，在较早的 DSP 芯片中没有这类中断。

软中断由程序指令产生，通常可分为三种情况：由中断指令引起的中断，由 CPU 的一些错误引起的中断以及为调试程序设置的中断。其中当 CPU 执行一条中断指令后，会立即产生中断，并且调用系统中相应的中断处理程序来完成中断功能。中断类型由中断指令的操作数指出。

由于中断源信号的种类很多，为了能够让 CPU 根据重要程度区别处理，C5000 系列的 CPU 为不同的中断设置了不同的优先级。其中根据能否被屏蔽，可以将中断分为三种类型，即复位（RESET）中断、不可屏蔽中断和可屏蔽中断。

（1）复位中断

复位中断具有最高优先等级，其相应信号为 RESET（$\overline{\text{RS}}$）信号，它可以停止 CPU 工作，并使之返回到一个默认的初始状态。RESET 信号是低电平有效信号，要能够使 DSP 正确复位，RESET 信号的低电平必须保持 10 个时钟脉冲。

（2）不可屏蔽中断

不可屏蔽中断为第二优先等级，相应信号为 $\overline{\text{NMI}}$ 信号。它通常用来向 CPU 发出一系列硬件问题的警报。DSP 对这类中断总是响应的，并从主程序转移到中断服务程序（Interrupt Service Routine，ISR)。除此 $\overline{\text{NMI}}$ 之外，所有软件中断也是不可屏蔽中断。

（3）可屏蔽中断

可屏蔽中断为第三优先等级。这些中断的优先级别比 $\overline{\text{NMI}}$ 中断和复位中断低。$\overline{\text{NMI}}$ 中断可以中止当前一个可屏蔽中断执行过程，而 $\overline{\text{NMI}}$ 或可屏蔽中断无法中止一个 $\overline{\text{NMI}}$ 的执行。

（4）中断优先级

C5000 系列 DSP 中，C54x 根据芯片型号不同，有 24~27 个软件和硬件中断，共分为 14 个中断优先级，可以实现多任务嵌套。C55x 根据芯片型号不同，有 32 个软件和硬件中断，共分为 27 个中断优先级。表 9-1 是 C5402 的中断优先级列表，表 9-2 是 C5509 的中断优先级列表。C5509 的这些中断名称 IVn 与外部中断、串口中断等相关联。

表 9-1　C5402 中断优先级列表

中断优先级	中断名称	中断优先级	中断名称
1（最高）	$\overline{\text{RS}}$ /SINTR	9	DMAC0 /SINT6
2	$\overline{\text{NMI}}$ /SINT16	10	TINT1/ DMAC1/SINT7
3	$\overline{\text{NMI0}}$ /SINT0	11	$\overline{\text{NMI3}}$ /SINT8
4	$\overline{\text{NMI1}}$ /SINT1	12	HPINT/SINT9
5	$\overline{\text{NMI2}}$ /SINT2	13	BRINT1/DMAC2/SINT10
6	TINT0/ SINT3	14	BXINT1/DMAC3/SINT11
7	BRINT0/ SINT4	15	DMAC4/SINT12
8	BXINT0/ SINT5	16（最低）	DMAC5/SINT13

表 9-2　C5509 中断优先级列表

中断优先级	中断名称	中断优先级	中断名称
1（最高）	RESETIV	14	IV9
2	NMIV	15	IV10
3	BERRIV	16	IV11
4	IV2	17	IV19
5	IV16	18	IV12
6	IV3	19	IV13
7	IV4	20	IV20
中断优先级	中断名称	中断优先级	中断名称
8	IV5	21	IV21
9	IV17	22	IV14
10	IV6	23	IV15
11	IV7	24	IV22
12	IV8	25	IV23
13	IV18	26（最低）	IV26

9.2.1.2 中断响应控制

当 CPU 检测到有效的中断源信号后，CPU 会自动停止当前指令的执行，转而去处理和该中断源相关的任务——中断服务程序(ISR)，这个过程称为中断响应。在这个过程中，CPU 会使用多个控制寄存器，并根据这些寄存器判断是否执行中断服务程序。

1．中断控制寄存器

当 CPU 检测到有效的可屏蔽中断源信号后，会在中断标志寄存器（Interrupt Flag Register，IFR）的相应比特位置上置 1。每个可屏蔽中断在 IFR 中都有一个对应的比特位，用于标志该种中断已经被检测到。图 9-1（a）是 C5402 的 IFR 比特位置。IFR 的比特位可以被 DSP 的中断器自动检测，也可以用软件来检测，以便进行相应的处理。

15	14	13	12	11	10	9	8
Rsvd	Rsvd	DMAC5	DMAC4	BXINT1	XRINT1	HINT	INT3
7	6	5	4	3	2	1	0
BXINTT2	BRINT2	BXINT0	BRINT0	TINT	INT2	INT1	INT0

（a）IFR

15	14	13	12	11	10	9	8
Rsvd	Rsvd	DMAC5	DMAC4	BXINT1	XRINT1	HINT	INT3
7	6	5	4	3	2	1	0
BXINTT2	BRINT2	BXINT0	BRINT0	TINT	INT2	INT1	INT0

（b） IMR

图 9-1　C5402 的中断标志寄存器（IFR）和中断屏蔽寄存器（IMR）

DSP 芯片为所有的可屏蔽中断提供了对应的控制寄存器——中断屏蔽寄存器（Interrupt Mask Register，IMR）。每个可屏蔽中断在 IMR 中都有一个对应的比特位，控制该种中断是否屏蔽。图 9-1（b）是 C5402 的 IMR 比特位置。当向任何一个 IMR 比特位写 1 时，都可以使能响应的中断；当向 IMR 比特位写 0 时，该中断被屏蔽。对 IMR 的寄存器处理可以按照下面的指令完成。

```
设置：  STM     #102h,IMR
修改：  ORM     #40h,*(IMR)
        ANDM    #0FFBFh,*(IMR)
```

除了 IMR 可以控制可屏蔽中断的响应之外，DSP 还提供了一个全局中断使能比特位 INTM（C54x 芯片中 ST1 的第 11 位，C55x 芯片中 ST1_55 的第 11 位）。当 INTM＝0 时，所有没有屏蔽的可屏蔽中断都被使能；当 INTM＝1 时，所有的可屏蔽中断都被禁止，DSP 不再响应这些中断。

2．中断响应条件

利用中断控制寄存器，DSP 可以根据需要来响应检测到的中断源信号。这些中断源对应的中断被响应的条件如图 9-2 所示。

（1）中断是否被屏蔽

这个条件可以通过检查 IMR 的相应比特位以及 INTM 来判断。

（2）中断是否被允许

这个条件可以通过检查 IFR 的比特位，比较中断优先级判断：当有高优先级中断发生时，

低优先级中断被阻止；当同时有不同的中断源向 CPU 发出中断请求时，CPU 响应级别最高的中断请求。

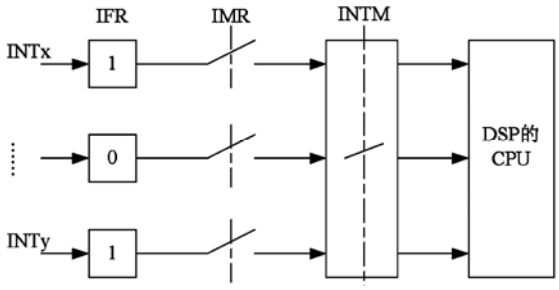


图 9-2　中断响应的屏蔽控制

9.2.1.3　中断矢量（Interrupt Vector）

DSP 芯片能够支持多种不同的中断，而每个中断都有自己的中断服务程序，代码长度不定。为了保证 CPU 能够及时调用中断服务程序，需要采用中断矢量来确定 CPU 响应每个中断时程序指针（PC）的跳转地址。

中断矢量有两种实现方法：固定的中断矢量和软件可重定位的中断矢量。80C51 系列的单片机采用的是固定中断矢量形式，其中断服务程序的跳转地址固定在 03h、0bh 等地址；而 TI 公司的 C5000，C6000 等系列 DSPs 芯片为了方便 DSP 存储器的配置，都采用了软件可重定位中断矢量形式。

C54x 系列的 DSP 为每个中断提供了一个中断矢量号，并在中断矢量表中提供了 4 个字的空间用于服务程序的存放。C54x 系列 DSP 有 32 个中断，中断矢量表占用 128 个字的空间，可以存放在除保留空间外的程序存储区任何位置。中断矢量表的基地址由中断矢量指针（Interrupt Vector Pointer，IPTR）比特位来确定，这些比特位对应于 PMST 的 15~7 位（如图 9-3 所示）。

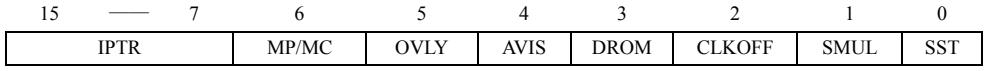


图 9-3　PMST 的比特位置

中断矢量的地址通过如下方式产生：

中断矢量地址 ＝(IPTR)<<11 ＋ (中断矢量序号)<<2

例如，当 IPTR = 1ffh 时，外部中断 $\overline{\text{INT0}}$ 被拉低。由于 $\overline{\text{INT0}}$ 对应的中断矢量号为 16，因此产生的中断矢量地址为#1ffh<<11+#10h<<2 = #0FFC0h（如图 9-4 所示）。

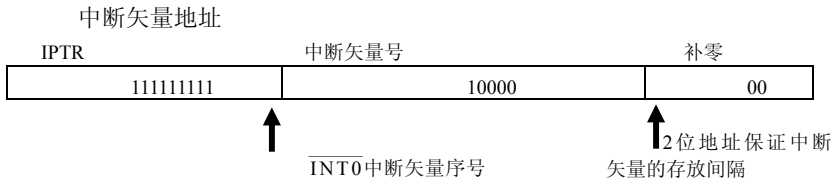


图 9-4　中断矢量地址产生示意图

9.2.1.4 中断处理过程

DSP 的 CPU 检测到有效的中断源信号后，将进行一系列的中断操作（如图 9-5 所示）。首先，CPU 判断中断响应条件是否满足。一旦满足条件，CPU 即开始响应该中断请求，执行 ISR。在 ISR 中，通常要保护 CPU 内部寄存器的值（保护现场）。在中断服务程序执行完后，还要恢复 CPU 内部寄存器的值（恢复现场）。

通常，CPU 在响应中断矢量号为 n 的中断请求时，自动完成如下操作：

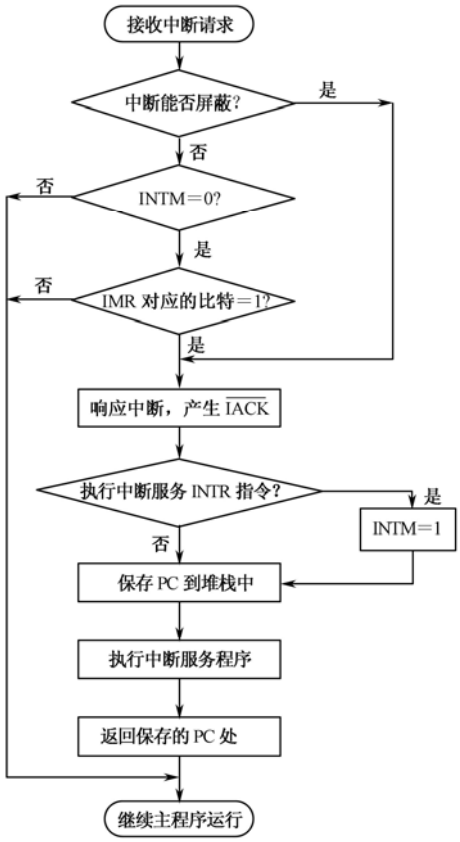


图 9-5　DSP 中断服务的处理流程

- $\overline{\text{INTM}}$ ＝1，全局中断禁止，防止其他中断嵌套；
- PC →*(SP) ，PC 值递减推入堆栈（保护返回地址）；
- vector(n) →PC，产生中断矢量地址加载 PC；
- $\overline{\text{IACK}}$ 拉低，表明中断发生；
- IFR(n)=0，清除该中断请求相应的标志位。

完成这些操作后，CPU 开始执行 ISR。

9.2.2 DSP 中断资源

下面以 C54x 系列 DSP 芯片 TMS320VC5410 为例简要介绍 DSP 芯片具体的中断资源，其他型号的芯片中断资源读者可以参考相应的芯片手册。

TMS320VC5410 DSP 芯片与中断相关的寄存器有 IMR（内存地址为 00h），IFR（内存地址为 01h）和 PMST（内存地址为 1dh）；复位时，IPTR= 1FFh，其中断矢量表的基址为 0xff80h；复位完成后，中断矢量表的地址可以通过修改 IPTR 实现重定位。例如，如果 DSP 的中断矢量表位于 0x0080h 处，则可以按如下方式修改 PMST。

```
ORM     #80h, PMST
ANDM    #007fh, PMST
```

C5410 的中断矢量如表 9-3 所示，其中可以通过软件屏蔽或放开的软、硬件中断有如下 18 个：4 个外部中断 $\overline{\text{INT}}$（0~3），1 个主机接口中断（HPINT），1 个定时中断（TINT），3 个缓冲接收串口（BRINT），3 个缓冲发送串口（BXINT），6 个 DMA 通道中断（DMACx）。有的中断共用同一个中断矢量号。

表 9-3　C5410 的中断资源

中断矢量号	中断名称	偏移量（Hex）	优先级	功能
0	$\overline{\text{RS}}$ /SINTR	00	1	复位（硬件或软件）
1	$\overline{\text{NMI}}$ /SINT16	04	2	非屏蔽中断
2	SINT17	08	—	软中断 SINT17
3	SINT18	0C	—	软中断 SINT18
4	SINT19	10	—	软中断 SINT19
5	SINT20	14	—	软中断 SINT20
6	SINT21	18	—	软中断 SINT21
7	SINT22	1C	—	软中断 SINT22
8	SINT23	20	—	软中断 SINT23
9	SINT24	24	—	软中断 SINT24
10	SINT25	28	—	软中断 SINT25
11	SINT26	2C	—	软中断 SINT26
12	SINT27	30	—	软中断 SINT27
13	SINT28	34	—	软中断 SINT28
14	SINT29	38	—	软中断 SINT29
15	SINT30	3C	—	软中断 SINT30
16	$\overline{\text{INT0}}$ /SINT0	40	3	外部中断 0，软中断 0
17	$\overline{\text{INT1}}$ /SINT1	44	4	外部中断 1，软中断 1
18	$\overline{\text{INT2}}$ /SINT2	48	5	外部中断 2，软中断 2
19	TINT/SINT3	4C	6	定时中断，软中断 3
20	BRINT0/SINT4	50	7	McBSP 0 接收中断，软中断 4
21	BXINT0/SINT5	54	8	McBSP 0 发送中断，软中断 5
22	BRINT2/DMAC0/SINT6	58	9	McBSP 2 接收中断（默认）/DMA 通道 0 中断，软中断 6
23	BXINT2/DMAC1/SINT7	5C	10	McBSP 2 发送中断（默认）/DMA 通道 1 中断，软中断 7
24	$\overline{\text{INT3}}$ /SINT8	60	11	外部中断 3，软中断 8
25	HPINT/SINT9	64	12	HPI 中断，软中断 9
26	BRINT1/DMAC2/SINT10	68	13	McBSP 1 接收中断/（默认）DMA 通道 2 中断，软中断 10
27	BXINT1/DMAC3/SINT11	6C	14	McBSP 1 发送中断（默认）/DMA 通道 3 中断，软中断 11
28	DMAC4/SINT12	70	15	DMA 通道 4 中断，软中断 12
29	DMAC5/SINT13	74	16	DMA 通道 5 中断，软中断 13
30	Reserved	78	—	保留
31	Reserved	7C	—	保留

9.2.3 DSP 中断程序实现

利用 DSP 中断资源进行系统开发的过程可以分为两大部分：中断初始化和中断服务。中断初始化过程主要包括中断引脚或信号的连接和选择、中断矢量表的建立、中断控制寄存器的设置（全局中断、单个中断）等。其中中断信号的连接和选择属于硬件设计工作，后面的章节将做介绍，其他的初始化过程可以在 DSP 复位完成后的程序段中实现。从图 9-5 中可知，中断服务过程需要用户编写 ISR 实现，对中断请求信号的检测、中断标志的设置等都是 CPU 自动完成的。由于在 C54x 中中断矢量表为每个中断保留了 4 个字的空间，因此对于较长的中断服务代码，只能通过在中断矢量表中进行跳转实现。

下面将给出 DSP 中断初始化、中断矢量表以及中断服务程序的例子。程序利用 DSP 的 $\overline{\text{INT0}}$ 引脚上检测信号波形，并通过中断程序在 XF 引脚上输出方波，频率是 $\overline{\text{INT0}}$ 输入波形频率的 1/2。

【例 9.1】中断初始化代码实例

```
INT_ini:    ANDM    #7fh,PMST      ; 对中断矢量表重定位
            ORM     #1180h,PMST
            STM     #0,IFR         ; 为 INT0 设置 IFR，IMR
            ORM     #1,IMR
            RSBX    INTM
            RET
```

【例 9.2】中断矢量表代码实例

```
.sect  "vectors"
RSV: BD    Reset
           STM #STK+LEN,SP
NMV:       RETE                   ; Put NMI routine here …
           NOP
           NOP
           NOP
           .space  64             ; 间隔 4 个字
           ...
INT0_I:    BD   INT0_isr          ; 间隔 4 个字，跳转到 INT0_isr 中
           PSHM    ST0
           PSHM    ST1
INT1_I:    BD   ISR1              ; 间隔 4 个字，跳转到 INT1_isr 中
           PSHM    ST0
           PSHM    ST1
           …
IVn:       BD   ISRn              ; 间隔 4 个字，跳转到 ISRn 中
           PSHM    ST0
           PSHM    ST1
           ...
```

【例 9.3】中断服务程序实例

```
        .mmregs
        .data
INT_Flag        .word  0
```

```
        .text
main:
        STM     81h,SP                  ; 初始化堆栈
        ADDM    #4,SP
        ANDM    0fffeh,SP
        CALL    INT_ini             ; 初始化中断 INT0 相关寄存器
Loop:
        B       Loop

INT0_isr:
        CMPM    *(INT_Flag),1
        BC      OUT_1,TC
        RSBX    XF
        ST      #1,*(INT_Flag)
        B       INT_END
OUT_1:
        SSBX    XF
        ST      #0,*(INT_Flag)
INT_End:
        POPM    ST1
POPM    ST0
RETE
```

一般而言，编写中断服务程序时应该注意如下几点：

（1）必须把在中断服务程序中使用到的寄存器值保存到堆栈中（保护现场），返回时以相反的次序恢复这些值（恢复现场）。对于 C5000 系列的 DSP 芯片来说，一般需要保护 ST0 和 ST1，因为其中保存了处理器状态的重要信息。

（2）如果在 ISR 中使用了 RPTB 指令，则必须保护 BRC 寄存器，而且必须在恢复 ST1 之前，恢复 BRC，否则先恢复的 ST1 中 BRAF 位会被清零。

（3）如果是与 C 语言混合编程，最好把 AR1、AR6 和 AR7 也保存起来，这是 C 编程默认的变量寄存器。

（4）如果利用 C 语言实现中断服务程序，只需要在函数名前使用关键字 interrupt 即可。此时实现的中断服务程序将自动地把 DSP 状态寄存器和辅助寄存器都进行保护。

9.3　DMA

信号处理中经常涉及大量的数据传输。利用 DSP 的指令可以完成数据的搬移，但这会占用大量的 CPU 资源。为降低 CPU 的负荷，通常都在 DSP 片内设计多通道的直接存储器访问（DMA）控制器。DMA 控制器是独立于 CPU 的设备，一旦正确初始化后，就能独立于 CPU 工作，在 CPU 操作的同时实现片内存储器、片内外设以及外围器件间的数据传输。

DMA 传送时，需要使用系统的地址和数据总线以及一些控制信号线，但这些总线一般都是由 CPU 控制的，因此为了能够实现 DMA，需要由硬件自动实现总线的控制权转移，为此一般的 DMA 控制器需要具有以下功能：

（1）可以向 CPU 发出 HOLD 信号，请求 CPU 让出总线，即 CPU 连在这些总线上的引线处于高阻状态；

（2）CPU 让出总线后，可以接管对总线的控制；

（3）可以在总线上进行寻址和读写控制；

（4）可以决定传送的数据个数；

（5）可以启动数据的传送，可以判断数据传送是否结束并发出结束信号；

（6）可以在结束传送后自动交出总线控制权，恢复 CPU 正常工作状态。

DSP 芯片提供了丰富的 DMA 资源，如在 C54x 系列 DSP 芯片中，DMA 控制器包含有六个独立的可编程通道，可以支持六个不同内容的 DMA 传输。同时，HPI 接口也可以使用 DMA 总线进行传输。C6000 系列的 DMA 控制器具有四个独立编程的传输通道，允许进行四个不同内容的 DMA 传输， C6000 系列芯片还有 EDMA 模块，可用于数据传输。因此，充分利用 DMA，可以更好地利用 DSP 的计算资源。本节将以 C54x 系列 DSP 芯片重点介绍 DSP 的 DMA 传输机制和配置使用方法。



9.3.1 DMA 数据传输模式

虽然不同的芯片中 DMA 控制器实现的功能有不同，但 DMA 控制器完成的基本操作都是读操作和写操作。其中在读操作中，DMA 控制器从存储空间中的源地址处读取数据，在写操作中，DMA 控制器将读操作中读取的数据写入到存储空间中的目的地址处。源地址和目的地址对应的存储空间可以为程序空间，数据空间和 I/O 空间。

为便于数据传输，在基本操作的基础上，DSP 芯片的 DMA 控制器将数据进行了分块（block）管理，每一块又可以分为若干帧（frame），每帧由一定长度的数据单元（element）组成。每个 DMA 通道的数据块和数据帧大小可以由用户自行设定。

基于这些管理单元，DMA 控制器提供了多种传输方式，包括元素传输、帧传输和块传输等。三种传输方式的最主要区别在于传输数据量的大小：

（1）元素传输只对一个数据进行读/写操作；

（2）帧传输将搬移一帧内的所有数据；

（3）块传输搬移块内所有帧的数据。

除了上述的一些特点外，DSP 芯片的 DMA 传输还可以根据需要确定数据的寻址方式。DMA 对每个通道的源和目的地址都提供了可配置的寻址方式，包括地址固定、地址增一、地址减一以及可编程的地址调整等。同时不同通道的 DMA 传输也可以设置不同的优先级，方便用户的选择。

为了实现数据的自动传输，DMA 控制器实现了与外部中断等事件的同步功能。相应的事件发生后，都将自动触发 DMA 传输，避免了用户对 DMA 过多的控制。同时 DMA 传输完毕后，也可以提供给 CPU 以中断，方便用户在数据传输后进行相应的处理。



9.3.2 DSP 的 DMA 控制

9.3.2.1 控制寄存器

和一般的外设控制相同，对 DMA 控制器的配置和操作可以通过设置一组存储器映射的控

制寄存器来实现。由于涉及的寄存器较多，为节省存储空间的地址占用量，C54x 系列 DSP 芯片采用了子地址寻址方式。

寄存器子地址寻址指的是在同一个地址指向的存储空间中实现多个寄存器的复用（如图 9-6 所示）。这时，需要使用一个子地址寻址寄存器来区分不同的寄存器，而这些寄存器内容的读写都通过一个子地址数据寄存器来完成。

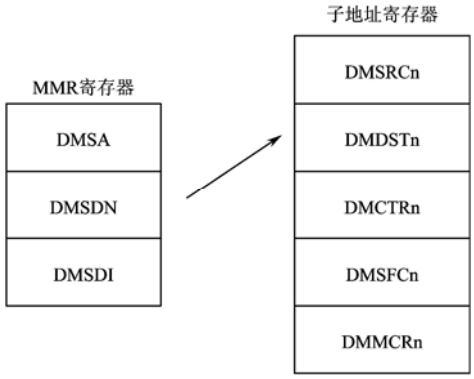


图 9-6　DMA 寄存器子地址寻址示意图

图 9-6 中，DMSA 是 DMA 子地址寻址寄存器（Subbank Address Register），DMSDI 是子地址自增 1 的 DMA 子地址数据访问寄存器（Subbank Access Register with Autoincrement），DMSDN 是不自增 1 的 DMA 子地址数据访问寄存器（Subbank Access Register without Autoincrement）。如果所有的 DMA 控制寄存器都要配置，使用 DMSDI 将非常方便。

为了访问一个特定的子地址寄存器，首先要向 DMSA 写入子地址，然后，才可以在物理空间中真正选定所要访问的寄存器 DMSDI 和 DMSDN。此时，对子地址数据寄存器的操作就是对选定的寄存器内容的操作。

由于 DMA 传输不需要 CPU 处理，因此在传输前需要提前确定好源地址、目的地址以及各自的修改方式、数据传输的宽度、数量以及传输时机等很多因素，这些因素都有对应的寄存器和控制比特。C54x 系列 DSP 芯片为每个 DMA 通道都独立设置了一系列的控制寄存器，如图 9-6 右侧所示，所有的 DMA 控制寄存器的定义如表 9-4 所列。

9.3.2.2　DMA传输控制

在启动 DMA 传输之前，需要对 DMA 通道进行初始化。在数据传输的过程中，DMA 控制器需要与 CPU 进行一定的事件交互，这需要通过中断或同步事件来完成。因此对 DMA 的控制重点是对 DMA 寄存器的配置以及 DMA 传输事件的管理。

1．全局控制初始化

通道优先级和使能控制寄存器 DMPREC 可以对 DMA 系统中的全局操作进行管理，其比特位置如图 9-7 所示，各比特域的功能在表 9-5 中列出。

根据系统需要，开发者可以独立地选择所使用的 DMA 通道并设置通道的优先级。向 DE[5:0]的第 n 位写 1 即使能通道 n，向 DPRC[5:0]的第 n 位写 1 即设置该通道为高优先级。当 DMA 的块传输结束时，DMA 控制器可以自动将使能比特复位。

表 9-4　C54xDSP 的 DMA 寄存器资源

名　　称	地　　址	子地址	作　　用
DMPREC	54h		通道优先级和使能控制寄存器
DMSA	55h		子地址寻址寄存器
DMSDI	56h		自增 1 子地址数据访问寄存器
DMSDN	57h		不自增 1 子地址数据访问寄存器
DMSRCn		n*5	DMA 通道 n 的源地址寄存器
DMDSTn		n*5+1	DMA 通道 n 的目的地址寄存器
DMCTRn		n*5+2	DMA 通道 n 的单元计数寄存器
DMSFCn		n*5+3	DMA 通道 n 的同步选择和帧计数寄存器
DMMCRn		n*5+4	DMA 通道 n 的传输模式控制寄存器
DMSRCP		30	DMA 源程序页地址
DMDSTP		31	DMA 目的程序页地址
DMIDX0		32	单元地址索引寄存器 0
DMIDX1		33	单元地址索引寄存器 1
DMFRI0		34	帧地址索引寄存器 0
DMFRI1		35	帧地址索引寄存器 1
DMGSA		36	全局源地址重载寄存器
DMGDA		37	全局目的地址重载寄存器
DMGCR		38	全局单元计数重载寄存器
DMGFR		39	全局帧计数重载寄存器

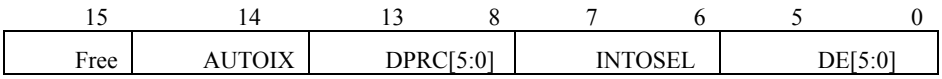


图 9-7　DMPREC 比特位置

表 9-5　DMPREC 的比特域功能说明

比特域	功　能
DE[5:0]	通道使能，第 n 位对应第 n 个 DMA 通道
INTOSEL	中断复用控制，可以选择不同的中断矢量表位置
DPRC[5:0]	通道优先级，第 n 位对应第 n 个 DMA 通道
AUTOIX	扩展重载方式
Free	调试控制位，=1 时自由运行，不受仿真器影响

当 DMA 传输结束时，DMA 控制器可以向 CPU 发出中断请求。为了利用有限的中断矢量实现多通道 DMA 中断，C54x 系列的 DSP 采用了中断复用实现方式，由 INTOSEL 控制。不同型号的 DSP，其中断复用的具体方法有所不同，如 C5410 的中断复用设置如表 9-6 所示。

表 9-6　C5410 的中断复用

中断序号	INTOSEL=00b	INTOSEL=01b	INTOSEL=10b	INTOSEL=11b
4	McBSP0 的 RINT	McBSP0 的 RINT	McBSP0 的 RINT	Reserved
5	McBSP0 的 XINT	McBSP0 的 XINT	McBSP0 的 XINT	Reserved
6	McBSP2 的 RINT	McBSP2 的 RINT	DMA 通道 0 中断	Reserved
7	McBSP2 的 XINT	McBSP2 的 XINT	DMA 通道 1 中断	Reserved
10	McBSP1 的 RINT	DMA 通道 2 中断	DMA 通道 2 中断	Reserved
11	McBSP1 的 XINT	DMA 通道 3 中断	DMA 通道 3 中断	Reserved
12	DMA 通道 4 中断	DMA 通道 4 中断	DMA 通道 4 中断	Reserved
13	DMA 通道 5 中断	DMA 通道 5 中断	DMA 通道 5 中断	Reserved

2．通道控制初始化

从表 9-4 中可以看到，每个 DMA 通道都有 1 组专用寄存器，这些寄存器共同控制该通道的 DMA 传输。其中，DMSFC 和 DMMCR 分别控制 DMA 通道的同步选择和工作模式，它们的比特位置和功能如表 9-7 所示。

表 9-7　DMSFC 和 DMMCR 的比特说明

常用寄存器	比特位置	比特域	控制功能描述
DMSFC	Bit15-12	DSYN[3:0]	DMA 同步事件
	Bit11	DBLW	双字模式，=0 时每个单元 16 比特，=1 时 32 比特
	Bit10-8	保留	
	Bit7-0	Frame Count	每块的帧计数，实际传输帧数为 Frame Count＋1
DMMCR	Bit15	AUTOINIT	自动初始化选择
	Bit14	DINM	DMA 中断生成屏蔽控制
	Bit13	IMOD	DMA 中断生成模式控制
	Bit12	CTMOD	DMA 传输计数模式控制，多帧模式/ABU 模式
	Bit11	SLAXS	源地址内部空间/外部空间访问选择
	Bit10-8	SIND	源地址索引变化模式
	Bit7-6	DMS	源地址空间选择
	Bit5	DLAXS	目的地址内部空间/外部空间访问选择
	Bit4-2	DIND	目的地址索引变化模式
	Bit1-0	DMD	目的地址空间选择

为了方便不同的外设使用 DMA 传输数据，DMA 控制器可以利用 DSYN 比特域确定用于在该通道初始化 DMA 传输的同步事件，例如可以选择 McBSP0 的接收事件、外部中断 3 事件等来触发 DMA 传输。DMMCR 寄存器 CTMOD 比特域可以控制 DMA 通道工作在不同的传输模式，可以使用的传输模式有两种：多帧模式和自动缓冲。一般情况下，DMA 工作在普通的多帧模式，当需要与缓冲接口相连时，可以采用自动缓冲模式与其匹配。同时，利用 AUTOINIT 也可以选择让 DMA 控制器在每次块传输完成后自动利用设置好的值进行初始化。

DMA 传输过程中，DMA 控制器从源地址中读取数据，并向目的地址中写数据，源地址和目的地址分别保存在 16 位寄存器中 DMSRCn 和 DMDSTn 中。当进行扩展寻址时，其低 16 位地址保存其中。这两个寄存器必须在 DMA 传输之前由软件设置，在传输过程中可由 DMA 控制器自动更新。

DMA 传输支持不同的地址空间。在 DMMCR 中提供了两个比特域 DMS 和 DMD，分别可以确定传输源地址和目的地址的空间属性，可以选择为程序空间、数据空间和 I/O 空间。另外，DMMCR 中的 SIND 和 DIND 比特域还规定了源地址和目的地址的索引更新方式：在每次传输结束后，地址索引可以根据需要选择为不修改、预先增 1、滞后增 1、滞后增加预设的索引值(这些索引值可以保存在 DMIDX0、DMIDX1，DMIDX0 和 DMFRI0，DMIDX1 和 DMFRI1）等。单元或帧索引寄存器（DMIDXn 和 DMFRIn）的值在源地址和目的地址的修改上都可以使用，而且这几个索引寄存器为整个 DMA 设定索引修改模式，并不区分通道。

在 DMA 传输前，还需要对 DMA 传输的数据个数进行设置。初始化的单元计数值保存在 16 位寄存器 DMCTRn 中，如果需要传输 N 个数据，则 DMCTRn 保存的数据为 N－1。在普通传输模式下，DMCTRn 可以在每个单元传输后自动减 1。当每帧的最后一个单元传输完毕，DMCTRn 会自动重载初始化值。而在 ABU 模式下，DMCTRn 中保存的是缓冲区的大小，在

传输时该值并不改变。

DMA 的块传输中，可以使用 DMSFC 寄存器中比特域 Frame Count 来规定每块中的帧数，实际传输的帧数等于 Frame Count 的值加 1。如果选择了自动初始化模式，那么当最后一帧传输完毕后，帧计数可以由 DMGFR 自动重载。

3．总线仲裁

在 DMA 传输过程中，不可避免需要占用数据总线。为了实现对 DMA 控制器、CPU 的总线仲裁，DSP 芯片提供了两个引脚$\overline{\text{HOLD}}$和$\overline{\text{HOLDA}}$：外界将 DSP 芯片的$\overline{\text{HOLD}}$引脚置为低电平时，DSP 芯片响应外部 DMA 请求，当 HOLD 应答线$\overline{\text{HOLDA}}$变低时，DSP 芯片就将外部总线置为高阻状态，将总线控制权交出，DSP 芯片停止运行，保持当前运行状态。当外界将$\overline{\text{HOLD}}$线变高时，总线控制权重新交给 DSP 芯片，DSP 芯片继续运行。

对TMS320C54x而言，有两种DMA模式，一种就是上面介绍的工作方式，当外部请求DMA时，DSP芯片停止内部运行（状态位HM＝1)，另一种是外部请求DMA时，DSP芯片交出外部总线控制权，但程序可以继续在片内运行（状态位HM＝0）。



9.3.3　DSP 的 DMA 配置举例

下面以 C54x 为例，介绍 DSP 的 DMA 机制。DSP 的 DMA 传输基本单元可以是单字或双字长，帧和块也可以作为传送的单位。当所有的通道都联合使用时，可以在 4 个 CPU 周期内实现 1 个 16 位字的传送。为了实现自动传输，我们需要分析 DMA 的控制条件，包括源/目的地址、传输优先级、事件同步、元素/帧计数、序号更新模式等。

DMA 可用的资源包括片内的 DARAM、SARAM，外部 RAM 以及一些常用外设，然而其中并不是所有的 DSP 都能够利用 DMA 进行外部 RAM 的访问，这些需要查阅相应的用户手册。

由于 DMA 的寄存器采用了子地址寻址方式，因此向 DMA 寄存器中写数据是一个多步骤的过程。首先需要向 DMSA 写入需要控制的寄存器子地址，然后根据需要对数据寄存器 DMSDN 或 DMSDI 写数进行配置。例如使用自动增 1 的子地址寻址模式时，对 DMA 寄存器的配置如下所示：

```
DMSRC0          .set        00h
    STM         DMSRC0,DMSA         ; 令 DMSA 指向 DMSRC0
    STM         #1000h,DMSDI        ; 向 DMSRC0 写入 1000h
    STM         #2000h,DMSDI        ; 向 DMDST0 写入 2000h
```

当采用事件同步和自动初始化选项时，有多种选项可以选择，例如从外部读入数据时，选择相应的同步事件为同步串口的接收满标志，可以利用该事件触发 DMA 传输，当 DMA 传输结束后向 DSP 的 CPU 发出中断信号。

下面的代码实例利用 DMA 将数据从程序空间搬移到数据空间，DMA 控制器配置为每个块进行 1000h 次字传输，源地址和目的地址在一个单元传输完成后都自增 1，整个传输自由运行。当第 1 个块传输完成后，DMA 通道利用全局重载寄存器的内容进行自动初始化，此时目的地址根据需要发生改变。

【例 9.4】DMA 传输代码实例

本例实现的要求如下：

（1）传输模式：多帧模式；

（2）初始化源地址：程序空间的 18000h；
（3）初始化目的地址：数据空间的 03000h；
（4）初始化传输块大小：1000h16 位字（1 帧，1000h 单元）；
（5）自动初始化源地址：程序存储空间的 18000h；
（6）自动初始化目的地址：数据存储空间的 02000h；
（7）自动初始化单元计数：1000h16 位字；
（8）自动初始化帧计数：000h (1 帧) ；
（9）事件同步：无；
（10）使用通道：DMA 通道 4。
实现的代码如下：

```
STM     DMSRCP,DMSA             ; 设置源地址页为 1
STM     #1h,DMSDN
STM     DMSRC4,DMSA       ; 设置源程序地址，由 DMSRCP 和 DMSRC4 共同构成，为 18000h
STM     #8000h,DMSDN
STM     DMDST4,DMSA             ; 设置目的地址为 3000h
STM     #3000h,DMSDN
STM     DMCTR4 ,DMSA            ; 设置传输单元个数为 1000h
STM      #(1000h–1) ,DMSDN
STM     DMSFC4 ,DMSA
STM     #0000000000000000b ,DMSDN   ;  0000~~~~~~~~~~~~~~ (DSYN)  没有同步事件
;  ~~~~0~~~~~~~~~~~~ (DBLW)  单字模式
;  ~~~~~000~~~~~~~~~ 保留
;  ~~~~~~~~00000000 (Frame Count)帧计数为 0（单帧）
STM     DMMCR4 ,DMSA
STM     #1000000100000101b ,DMSDN ;  1~~~~~~~~~~~~~~~~~ (AUTOINIT)自动初始化使能
;  ~0~~~~~~~~~~~~~~~ (DINM)中断屏蔽
;  ~~0~~~~~~~~~~~~~~ (IMOD) N/A
;  ~~~~0~~~~~~~~~~~~ (CTMOD)  多帧模式
;  ~~~~~0~~~~~~~~~~~ 保留
;  ~~~~~001~~~~~~~~ (SIND)  源地址后增 1
;  ~~~~~~~~00~~~~~~ (DMS)  源地址空间，程序空间
;  ~~~~~~~~~~0~~~~~ 保留
;  ~~~~~~~~~~~001~~ (DIND)  目的地址后增 1
;  ~~~~~~~~~~~~~~~01 (DMD)  目的地址空间，数据空间
STM     DMGSA,DMSA                  ; 设置全局源地址 8000h（程序空间 18000h 的低 16 位）
STM     #8000h,DMSDN
STM     DMGDA,DMSA                  ; 设置全局目的地址 2000h
STM     #2000h,DMSDN
STM     DMGCR,DMSA                  ; 设置全局的搬移单元数，1000h 个字
STM     #(1000h–1),DMSDN
STM     DMGFR,DMSA                  ; 设置全局帧计数，1 帧
STM     #0000000000000000b,DMSDN
STM     #0001000000010000b ,DMPREC ; 0~~~~~~~~~~~~~~~~~ (FREE) DMA 受仿真器影响
; ~0~~~~~~~~~~~~~~~ 保留
; ~~010000~~~~~~~~ (DPRC[5:0])通道 4 高优先级
```

```
; ~~~~~~~~00~~~~~~ (INTOSEL) N/A
; ~~~~~~~~~~010000 (DE[5:0]) 通道 4 使能，其他通道禁止
```

9.4 时序模块

DSP 芯片是一种高速并行执行的芯片，其片内的各组成模块对时序具有较高的要求，因此其片内集成的时序控制模块是系统设计时需要考虑的一个重要部分。TI 公司的各种系列 DSP 芯片上都集成有 PLL 时钟振荡模块、定时器模块以及总线时序模块等，本节将重点对它们进行介绍。

9.4.1 PLL 模块

早期的 DSP 芯片一般工作频率较低，时钟产生模块只需要直接利用输入时钟信号即可为片内产生稳定的工作时钟。随着 DSP 芯片速度的提高，如果仍然采用这种方式，则势必要求外部频率很高，而这很可能会给系统的稳定性带来影响。因此，现在的 DSP 芯片一般提供多种工作方式，包括分频方式和倍频方式，为方便使用，C5000 系列的 DSP 芯片还采用了更加灵活的可编程 PLL 方式。

9.4.1.1 时钟输入

给 DSP 芯片提供时钟一般有两种方法。一种是利用 DSP 芯片内部所提供的晶振电路，在 DSP 芯片的 X1 和 X2/CLKIN 之间连接晶体可启动内部振荡器，如图 9-8 所示。晶体应为基本模式，且为并联谐振。这种方式由于晶体没有电压问题，因此可以适应于任何 DSP。当系统中只有单一时钟信号时，这种方法使用较多。

另一种方法是将外部时钟源直接输入 X2/CLKIN 引脚，X1 悬空。采用封装好的晶体振荡器，这种方法使用方便，信号也比较稳定，因而得到了广泛的应用。有源晶振工作时，只要在 4 脚上加上 Vcc，2 脚接地，就可在 3 脚上得到所需的时钟，如图 9-9 所示。图中所画的晶体振荡器为其顶视图，其中的 1 脚悬空。利用这种方法可以根据需要同时为多个芯片提供同频时钟。

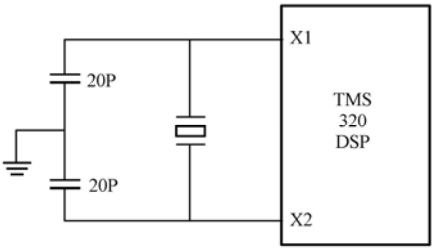


图 9-8 内部振荡电路

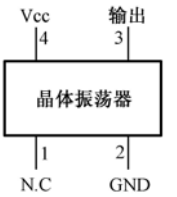


图 9-9 晶体振荡电路

9.4.1.2 PLL配置

传统的时钟产生模块一般都具有分频工作方式，其工作频率与外部提供的频率相等或者是外部频率的 2 分频或 4 分频。例如 TMS320C25-40，其指令周期是 100 ns，工作频率是 10MHz，外部提供 40 MHz，工作频率是外部频率的四分之一。为了利用较低的外部频率产生片内高频时钟，

C5000 系列 DSP 采用了可编程 PLL 方式。下面以 TMS320VC5402 为例介绍其时钟产生方式。

TMS320VC5402 内部具有一个可编程锁相环（PLL），受一个存储器映射（地址为 58h）的时钟模式（Clock Mode，CLKMD）寄存器控制（CLKMD 的比特定义如图 9-10 和表 9-8 所示）。VC5402 的 PLL 可以配置为以下两种模式（配置方法如表 9-9 所示）：

- PLL 模式：输入时钟乘以一个 1~31 之间的常数；
- DIV 模式：输入时钟除以 2 或 4。

15－12	11	10－3	2	1	0
PLLMUL	PLLDIV	PLLCOUNT	PLLON/OFF	PLLNDIV	PLLSTATUS

图 9-10　CLKMD 寄存器的比特定义

表 9-8　CLKMD 的比特位置和作用

比特位置	比特域	作　　用
15－12	PLLMUL	PLL 倍频器
11	PLLDIV	PLL 分频器
10－3	PLLCOUNT	PLL 计数值，用于设置 PLL 电路状态变化时的过渡时间长度
2	PLLON/OFF	PLL 使用/禁止控制位，和 PLLNDIV 共同控制 PLL 工作状态
1	PLLNDIV	PLL 时钟产生选择控制位，=0 时 DIV 模式，=1 时 PLL 模式
0	PLLSTATUS	PLL 状态指示位，=0 时 DIV 模式，=1 时 PLL 模式

表 9-9　CLKMD 的 PLL 状态配置

PLLON/OFF	PLLNDIV	PLL 状态
0	0	禁止
0	1	使能
1	0	使能
1	1	使能

当 DSP 处于 DIV 模式（PLLSTATUS=0）时，PLLMUL，PLLDIV，PLLCOUNT 和 PLLON/OFF 都不再起作用。当 DSP 处于 PLL 模式时，芯片工作时钟频率是输入时钟频率的倍频，倍频数受 PLLMUL 控制。DSP 片内时钟频率和输入时钟的关系如表 9-10 所示。

表 9-10　PLL 的倍频配置

PLLNDIV	PLLDIV	PLLMUL	倍频系数
0	X	0－14	0.5
0	X	15	0.25
1	0	0－14	PLLMUL +1
1	0	15	1
1	1	0 或偶数	(PLLMUL+1)/2
1	1	奇数	PLLMUL/4

除了软件配置之外，CLKMD 的复位值还可以根据 DSP 芯片三根输入引脚 CLKMD1~CLKMD3 确定。表 9-11 是 DSP 复位时钟模式设定，从表中可以看出，不同的 CLKMD1~CLKMD3 的值对应于相应的时钟模式。例如，设 CLKMD1~CLKMD3=001，则时钟模式是 PLL*10，如果外部的晶体频率是 10 MHz，则复位后 DSP 的工作频率是 10*10=100 MHz。

表 9-11　TMS320VC5402　DSP 复位时钟模式

CLKMD1	CLKMD2	CLKMD3	CLKMD 的复位值	时钟模式
0	0	0	E007h	PLL*15
0	0	1	9007h	PLL*10
0	1	0	4007h	PLL*5
1	0	0	1007h	PLL*2
1	1	0	F007h	PLL*1
1	1	1	0000h	1/2（PLL 无效）
1	0	1	F000h	1/4（PLL 无效）
0	1	1	—	保留

一般地，DSP 的程序需要从外部低速 EPROM 中调入，可以采用较低工作频率的 DSP 复位时钟模式，待程序全部调入到内部快速 RAM 后，再用软件重新配置 CLKMD 的值，使芯片工作在较高的频率上。例如，设外部晶体频率是 10MHz，设置 CLKMD1~CLKMD3=111，则复位后 DSP 的工作频率是 10MHz/2=5MHz，用软件重新设置 CLKMD=9007h，就可以使 DSP 工作在 10*10MHz=100MHz 的频率。

9.4.1.3　PLL配置举例

DSP 的时钟生成模块具有两种工作方式，因此 PLL 的配置存在两种情况，即由 DIV 模式切换到 PLL 模式和 PLL 模式切换到 DIV 模式，不管何种情况，都要保证 PLL 模块能够锁定。因此需要对 CLKMD 的状态进行检测，例 9.5 是 PLL 配置的软件实例，配置后工作频率是输入时钟的 10 倍。

【例 9.5】PLL 倍频的配置代码

```
.mmregs
        STM     #0,CLKMD
clk_chg:
        LDM     CLKMD,A
        AND     #01b,A
        BC      clk_chg,ANEQ        ; 检测 PLL 模块是否稳定
        STM     #5207h,CLKMD        ; PLLSTATUS=1,PLLNDIV=1,PLLON/OFF=1
```

9.4.2　定时模块

为了提供准确的定时信号，各种 DSP 芯片都在片内集成了定时模块，本节主要介绍 C54x 系列 DSP 片内的定时模块，包括其功能、控制和主要应用。

9.4.2.1　定时模块配置

C54x DSP 芯片片内定时器是一个软件可编程的计数器，它包括以下 3 个存储器映射寄存器，长度都是 16 位。

- 定时寄存器（TIM）：位于数据空间的 0024h 单元。其中数据进行自减操作。
- 定时周期寄存器（PRD）：位于数据空间的 0025h 单元。每次复位或 TINT 中断时将内容装入 TIM 寄存器。
- 定时控制寄存器（TCR）：位于数据空间的 0026h 单元，包含了定时器的控制位和状

态位。TCR 的比特分配和说明分别如图 9-11 和表 9-12 所示。

15-12	11	10	9-6	5	4	3-0
Reserved	Soft	Free	PSC	TRB	TSS	TDDR

图 9-11　TCR 的比特分配

表 9-12　TCR 比特位说明

比特位置	名比特域	说　明
15~12	—	保留
11	Soft	与 Free 位一起确定定时器状态；在 HLL 调试中若遇到断点，在 Free=0 时，Soft 选择定时器模式；Soft=0，定时器立即停止；Soft=1，计数器减到 0 后定时器停止
10	Free	与 Soft 位一起确定定时器状态；Free=0，选择 Soft 所确定的定时器模式；Free=1，Soft 制定模式无效，定时器运行
9~6	PSC	定时器预定标计数器。定义片内定时器的数目；当 PSC 减到 0 或复位时，PSC 内装入 TDDR 值
5	TRB	定时器重置；复位片内定时器
4	TSS	定时器停止状态；停止或者启动片内定时器；复位时，TSS=0 启动定时器，TSS=1 停止定时器
3~0	TDDR	定时器降周期因子；用于调整片内定时器自减到 0 的周期间隔；当 PSC 自减到 0 后，PSC 内载入 TDDR 值

片内定时器中，4 位的预定标计数器 PSC 和 16 位定时计数器 TIM 组成一个 20 位的计数器，定时器每个 CPU 时钟周期减 1，每次计数器减到 0 将产生定时器中断（TINT），同时 PSC 和 TIM 重新载入预设的值。

图 9-12 给出了定时器的方框图，它分为主定时模块和预定标模块。主定时模块包括 PRD 和 TIM，由预定标模块定时；预定标模块包括 TCR 中的 TDDR 和 PSC 比特，由 CPU 时钟定时。

在主定时模块中，对应于预定标模块的每个输出时钟，TIM 值减 1。当 TIM 减到 0 后，TIM 内载入 PRD 的值。当设备复位（$\overline{\text{SRESET}}$ =1）或者定时器单独复位（TRB=1）时，PRD 的内容也被载入 TIM。主定时模块的定时器中断（TINT）信号输出被送往 CPU 以及定时器的输出引脚 TOUT。

预定标模块中的 TDDR 和 PSC 均位于寄存器 TCR 中，每来一个 CPU 时钟，PSC 值减 1。与主定时模块一样，重置 PSC 值的时刻在 PSC 减到 0、设备复位或者定时器单独复位时，此时 TDDR 的内容复制到 PSC 中。

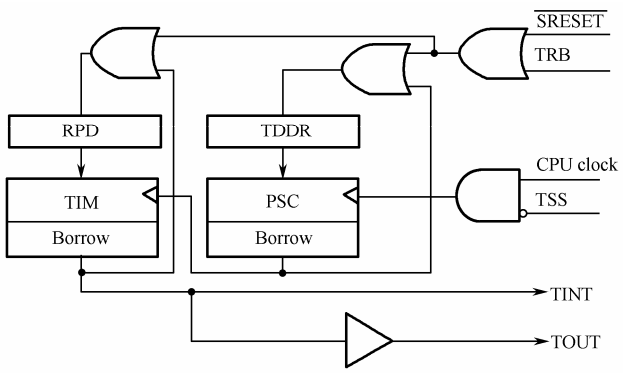


图 9-12　定时器方框图

定时器中断 TINT 的速率可由下面公式计算：

$$\text{TINT rate} = \frac{1}{t_c \times u \times v} = \frac{1}{t_c \times (\text{TDDR}+1) \times (\text{PRD}+1)}$$

式中 t_c 为 CPU 时钟周期，u 和 v 分别是 TDDR 和 PRD 设定的分频比。

9.4.2.2 定时模块配置举例

定时器的使用非常广泛。例如，使用 TOUT 引脚输出定时信号，可以产生模拟接口的采样时钟，对于芯片运行程序中的一些周期性任务，也需要利用定时中断来触发执行。

定时模块的使用关键是要对其进行正确的初始化。初始化定时器时，需要按照以下步骤设置：

（1）TSS=1，停止定时器；

（2）载入 PRD 值；

（3）重新载入 TCR 初始化 TDDR，设置 TSS=0 和 TRB=1 来重载定时器周期，启动定时器。

在状态寄存器 ST1 的中断模式 INTM=1 的情况下，可以采取下列步骤激活定时器中断：

（1）设置中断标志寄存器（IFR）中的 TINT=1，清除已有定时器中断；

（2）设置中断屏蔽寄存器（IMR）中的 TINT=1，使能定时器中断；

（3）设置 INTM=0，使能全部中断。

【例 9.6】下面是对定时器进行初始化的实例。

C54x DSP 芯片的工作时钟为 100 MHz，要求产生一个 100 μs 的定时中断，利用这个固定时间周期进行外部信号检测和处理。此时分频比为 10 000，因此可以分别对 TDDR 和 PRD 赋值为 3 和 2499。例程中没有给出中断服务程序代码。

```
TIMER_ini:
STM     #10h,TCR        ; 停止定时器
NOP
STM     #2499,PRD       ; 对 PRD 赋值
ORM     #23h,TCR        ; 对 TDDR 赋值，重置定时器
NOP
NOP
ANDM    #0ffefh,TCR     ; 启动定时器
```

9.4.3 等待状态发生器

DSP 的指令周期较快，在访问慢速存储器或外设时，为了保证芯片之间时序配合，通常需要插入等待状态。等待分硬件等待和软件等待两种方式，每一个系列 DSP 芯片实现的等待方式不完全相同。

（1）C2000 系列：硬件等待信号为 READY，高电平时不等待。软件等待由 WSGR 寄存器决定，可以加入最多 7 个等待，其中程序存储器和数据存储器及 I/O 可以分别设置。

（2）C3x 系列：硬件等待信号为/RDY，低电平时不等待。软件等待由总线控制寄存器中的 SWW 和 WTCNY 决定，可以加入最多 7 个等待，但等待是不分段的，除了片内之外全空间有效。

（3）C5000 系列：硬件等待信号为 READY，高电平时不等待。软件等待由 SWWCR 和 SWWSR 寄存器决定，可以加入最多 14 个等待，其中程序存储器、控制程序存储器和数据存

储器及 I/O 可以分别设置。

（4）C6000 系列（只限于非同步存储器或外设）：硬件等待信号为 ARDY，高电平时不等待。软件等待由外部存储器接口控制寄存器决定，总线访问外部存储器或设备的时序可以设置，可以方便地同异步存储器或外设接口。

9.4.3.1　硬件等待

早期的 DSP 芯片内部没有集成等待状态发生器，因此需要在外部用硬件来实现。在某些特殊场合，DSP 的软件等待无法满足需要时，也要使用硬件等待。这里我们以 TMS320C2x 为例来说明。TMS320C2x 支持与慢速外设接口的硬件等待状态插入。当与慢速外设接口时，系统必须提供能产生等待状态的硬件电路。外设存取的速度越慢，则所需插入的等待状态数就越多，那么，如何根据外设存取速度来确定所需的等待状态数呢？

设 t_a 为外设访问时间，则外设访问时间与等待状态数的关系为：

TMS320C25-40: [100(N−1)+40]ns<t_a<[100N+40]ns

TMS320C25-50: [80(N−1)+29]ns<t_a<[80N+29]ns

表 9-13 示出了两种 TMS320C25 芯片对应的等待状态数。

表 9-13　寻址时间与等待状态数的关系

等待状态数	TMS320C25-40	TMS320C25-50
0	t_a<40ns	t_a<29ns
1	40ns<t_a<140ns	29ns<t_a<109ns
2	140ns<t_a<240ns	109ns<t_a<189ns
3	240ns<t_a<340ns	189ns<t_a<269ns
4	340ns<t_a<440ns	269ns<t_a<349ns

如果系统中所有器件均为高速且不需要等待状态，则只要将 READY 线通过一上拉电阻接至高电平；若系统中所有外设均需插入一个等待状态，则仅需将$\overline{\text{MSC}}$线连接至 READY 中即可。同样，$\overline{\text{MSC}}$与$\overline{\text{PS}}$、$\overline{\text{DS}}$或$\overline{\text{IS}}$组合可对相应的部分插入一个等待状态，而其余部分仍以零等待运行。

9.4.3.2　软件等待

显然，采用硬件插入等待的方式是比较麻烦的。现在的 DSP 芯片一般在内部都设有等待机制，只需通过软件设置即可，可以省去外加硬件。下面以 TMS320VC5402 为例来说明。

TMS320VC5402 的可编程软件等待状态插入由软件等待状态寄存器（SWWSR）和软件等待状态控制寄存器（SWCR）控制。SWWSR 控制等待状态产生器的操作，各比特域的名称及功能如表 9-14 所示。它可以将外部总线周期扩展到 7 个机器周期（C549、C5402、C5410 和 C5420 位 14 个机器周期），方便与慢速的片内存储器和 I/O 器件接口。

C5402 DSP 片内 64 K 字的程序存储器和数据存储器都被划分为 2 个 32 K 字的块，加上 I/O 空间的 1 个 64 K 字的块，因此 SWWSR 中有 5 个 3 位字段分别对应于这 5 个块，通过设置 0~7 来控制这 5 个块的等待状态。

SWCR 是等待状态的倍增因子，仅最低位（SWSM）起作用，当该位为 0 时，倍增因子是 1，此时等待状态数仍为 0~7 个，当该位为 1 时，倍增因子是 2，此时等待状态数为 0~14 个。SWSM 的复位值为 0。

表 9-14　SWWSR 比特域名称及功能

比特位置	比特域名称	复位值	功　能
15	XPA	0	扩展程序地址控制位；XPA 与程序空间域（比特 5~0）一起决定等待状态的程序空间
14~12	I/O	1	I/O 空间；域值 0~7 对应于 0~7 个等待状态，作用范围是 0000~FFFFh 的 I/O 空间
11~9	Data	1	高端数据空间；域值 0~7 对应于 0~7 个等待状态，作用范围是 8000~FFFFh 的外部数据空间
8~6	Data	1	低端数据空间；域值 0~7 对应于 0~7 个等待状态，作用范围是 0000~7FFFh 的外部数据空间
5~3	Program	1	高端程序空间；域值 0~7 对应于 0~7 个等待状态，作用范围是：XPA=0 时为 x8000~xFFFFh 的外部程序空间，XPA=1 时不起作用
2~0	Program	1	程序空间；域值 0~7 对应于 0~7 个等待状态，作用范围是：XPA=0 时为 x0000~x7FFFH 的外部程序空间，XPA=1 时为 00000~FFFFFh 的外部程序空间

9.5　接口模块

接口（interface）是 DSP 等处理器的 CPU 或工作模块与外部设备之间的连接部件或电路，它是进行信息交换的通道。DSP 只有通过接口才能和外部设备进行变换和中转。随着微电子技术的发展，出现了各式各样的器件接口类型，为了便于采用这些常用接口与其他外部设备相连，DSP 芯片提供丰富的片内集成外设（peripherals onchip）实现接口功能，例如串口、主机接口、扩展存储器接口、通用 I/O 口等。本节将分析常用片内外设的工作原理和使用方法。

9.5.1　概述

一般而言，DSP 芯片内 CPU 的数据都是以三总线（数据总线、地址总线和控制总线）形式传送的，电气性能是固定的，负载能力较弱。在与接口设备相连接的时候，需要采用缓冲（提高负载能力）、锁存（保持数据）、降速（片内工作时钟相对于外部设备而言是极快的）等多种手段，实现电气特性、工作速度等特性的匹配。因此，DSP 片内外设接口主要实现如下功能：

（1）数据缓冲

为解决芯片 CPU 高速和外设低速的矛盾，避免因速度不一致而丢失数据，接口中一般都设置数据寄存器或锁存器。同时，接口电路还要提供寄存器“空”，“满”，“准备好”，或“忙”等状态信号，以便查询。

（2）接收和执行 CPU 命令

对于片内外设的控制命令，芯片的 CPU 一般以代码的形式发到接口的命令寄存器上，再由接口电路对命令代码进行分析，并传送给片内外设产生相应的操作。

（3）中断管理

当外设需要及时得到 DSP 芯片 CPU 的处理时，需要接口能够提供中断服务，因此接口中设置中断控制器，为芯片的 CPU 处理有关中断事务。

由于外设所需的控制信号和它提供的状态信号往往并不能和 DSP 芯片的总线信号匹配，因此需要信号转换，这种转换包括电平、电流以及时序的转换。除此而外，芯片内部处理的是并行数据，而有的外设需要进行串行传输，因此还需要进行“并/串”或“串/并”转换。由于 DSP 是数字器件，它的输入和输出数据都是若干位的数字量，有时还需要通过 A/D、D/A 反映成模拟量。因此，信号转换是接口设计中的一项重要任务。

接口的设计是硬件和软件相结合的实现过程。在开发 DSP 应用系统的时候，由于外部设

备种类很多，它们的电气性能和工作速度差别很大，虽然 DSP 芯片集成了外设接口，但同样还有其他一些实现方式可以选择，同时每种外设接口的使用方式也可以灵活选择。因此 DSP 系统开发的一个主要任务就是研究如何采用硬件和软件结合的方法，利用这些外设接口与外部设备进行最佳的连接和匹配，以保证高效、可靠的数据交换。

9.5.2　多通道缓冲串行口

串行口是 DSP 芯片上一种十分重要的外设接口。TMS320C54x 最初有三种类型的串行口：标准同步串行口、缓冲串行口（BSP）和时分复用串行口（TDM）。目前串行口主要以多通道缓冲串行口（McBSP）的实现形式为主，McBSP 可以完成标准串口的全双工串行通信，同时还能够支持多种协议下的直接接口。本节将以 C54x 系列 DSP 芯片详细介绍 McBSP 的资源和配置方法。

9.5.2.1　概述

C54x 的 McBSP 是在缓冲串口的基础上发展起来的。由于利用 McBSP 可以实现同步串口、缓冲串口的功能，因此在新推出的 DSP 上都只集成了 McBSP 接口。McBSP 主要具有如下特点：

- 具有双缓冲数据寄存器，允许连续的数据流，可以实现全双工通信；
- 支持多种方式的传输接口（包括 T1/E1 帧协议、AC97 兼容设备、SPI 设备等），可与工业标准的编/解码器、AICs（模拟接口芯片）以及其他串行 A/D、D/A 接口；
- 数据传输的时钟可以利用外部时钟，也可是片内的可编程时钟；
- 帧同步信号和时钟信号收发独立、极性可设，可编程程度高；
- 帧长度可以在 1~128 个字间选择，支持传输的数据字长可以是 8 bit 、12 bit、16 bit、20 bit、24 bit 或 32 bit；
- 支持多通道传输，可与多达 128 个通道进行收发；
- 内置 μ 律和 A 律硬件压扩；
- 当利用 DMA 为 McBSP 服务时，串行口数据读写具有自动缓冲能力。

McBSP 的组成框图如图 9-13 所示。

从图 9-13 中可以看到，McBSP 在结构上可以分为一个数据通道和一个控制通道。其中数据通道完成数据的发送和接收。CPU 或 DMA 控制器向数据发送寄存器（DXR）中写入待发送的数据，从数据接收寄存器（DRR）读取接收到的数据。写入 DXR 的数据通过发送移位寄存器（XSR）移位输出至 DX 引脚。同样，DR 引脚上接收到的数据先移位进入接收转换寄存器（RSR）中，然后被复制到接收缓冲寄存器（RBR）中，RBR 再将数据复制到 DRR 中，最后等候 CPU 或 DMA 控制器将数据读走。可见，数据传送是双缓冲机制，而接收是三级缓冲，这种多级缓冲方式使得片内的数据搬移和外部数据的通信可以同时进行。

McBSP 的通道数可达 128 个。它的收发时钟和帧定时是独立的。时钟可以是外部输入，也可以是内部产生的。不论是帧同步还是数据块的极性都可以编程控制，因此 McBSP 的数据传输能力很大。

控制通道完成的任务包括内部时钟产生、帧同步信号产生、对这些信号的控制以及多通道的选择等。为了能够自动传输，McBSP 提供了向 CPU 发中断和向 DMA 控制器发 DMA 事件的能力，这些也由控制通道负责完成（如表 9-15 所示）。

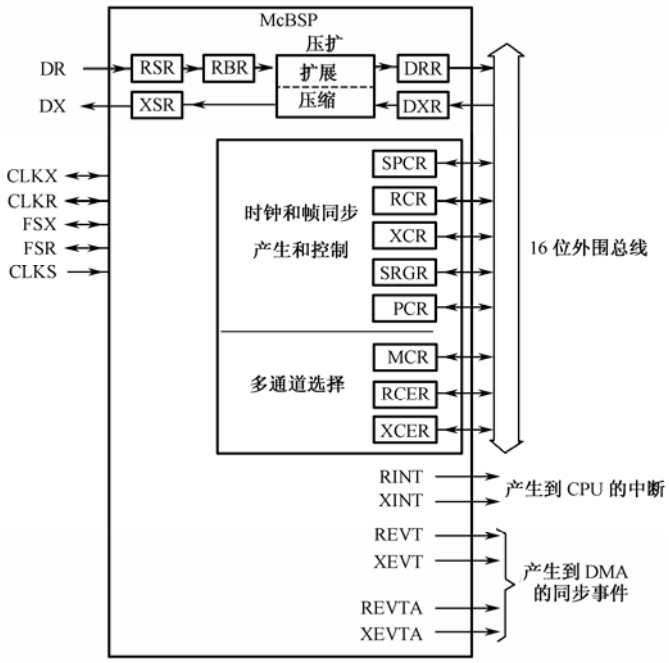


图 9-13　McBSP 组成框图

McBSP 的引脚和标准串口的定义相同。C54x 通过片内的外设总线访问串行口的控制寄存器，进而实现与 McBSP 间的通信与控制。McBSP 最高的比特率可以达到 CPU 时钟的一半。

表 9-15　McBSP 的 CPU 中断和 DMA 同步事件

信　　号	功　　能
RINT	接收中断，送往 CPU
XINT	发送中断，送往 CPU
REVT	接收同步事件，送往 DMA 控制器
XEVT	发送同步事件，送往 DMA 控制器

9.5.2.2　McBSP寄存器

McBSP 通过一系列存储器映射控制寄存器来进行配置和操作。由于使用的寄存器较多（如图 9-13 所示，包括 SPCR、RCR、XCR 等），因此和 DMA 控制器相同，也采用子地址寻址方案。McBSP 通过复接器将一组子地址寄存器复接到存储器映射的一个位置上。复接器由子块地址寄存器 SPSAx 控制。子块数据寄存器 SPSDx 用于指定子地址寄存器中数据的读写。其内部连接方式如图 9-14 所示。

为访问某个指定的子地址寄存器，首先要将相应的子地址写入 SPSAx，SPSAx 驱动复接器，使其与 SPSDx 相连，接入相应子地址寄存器所在的实际物理存储位置。当向 SPSDx 写入数据时，数据送入前面子地址寄存器中所指定的内嵌数据寄存器；当从 SPSDx 读取数据时，也接入前面子地址寄存器中所指定的内嵌数据寄存器。

TMS320C54x 中主要的 McBSP 控制寄存器如表 9-16 和表 9-17 所列。

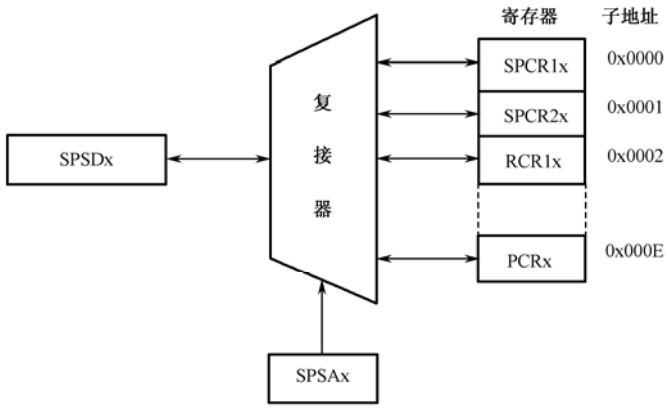


图 9-14　子地址映射示意图

表 9-16　C54x DSP 中的 McBSP 控制寄存器

寄存器名称	地　　址	说　　明
DRR20	0x20	McBSP0 的数据接收寄存器 2
DRR10	0x21	McBSP0 的数据接收寄存器 1
DXR20	0x22	McBSP0 的数据发送寄存器 2
DXR10	0x23	McBSP0 的数据发送寄存器 1
DRR22	0x30	McBSP2 的数据接收寄存器 2
DRR12	0x31	McBSP2 的数据接收寄存器 1
DXR22	0x32	McBSP2 的数据发送寄存器 2
DXR12	0x33	McBSP2 的数据发送寄存器 1
SPSA2	0x34	McBSP2 的子地址寻址寄存器
SPSD2	0x35	McBSP2 的子地址数据寄存器
SPSA0	0x38	McBSP0 的子地址寻址寄存器
SPSD0	0x39	McBSP0 的子地址数据寄存器
DRR21	0x40	McBSP1 的数据接收寄存器 2
DRR11	0x41	McBSP1 的数据接收寄存器 1
DXR21	0x42	McBSP1 的数据发送寄存器 2
DXR11	0x43	McBSP1 的数据发送寄存器 1
SPSA1	0x48	McBSP1 的子地址寻址寄存器
SPSD1	0x49	McBSP1 的子地址数据寄存器

表 9-17　C54x DSP 中子地址寻址的 McBSP 控制寄存器（x＝0,1,2）

寄存器名称	子地址	说　　明
SPCR1x	0x0000	McBSP 串口控制寄存器 1
SPCR2x	0x0001	McBSP 串口控制寄存器 2
RCR1x	0x0002	McBSP 接收控制寄存器 1
RCR2x	0x0003	McBSP 接收控制寄存器 2
XCR1x	0x0004	McBSP 传送控制寄存器 1
XCR2x	0x0005	McBSP 传送控制寄存器 2
SRGR1x	0x0006	McBSP 采样率生成寄存器 1
SRGR2x	0x0007	McBSP 采样率生成寄存器 2
MCR1x	0x0008	McBSP 多通道寄存器 1
MCR2x	0x0009	McBSP 多通道寄存器 2

（续表）

寄存器名称	子地址	说　明
RCERAx	0x000A	McBSP 接收通道使能寄存器——A 部分
RCERBx	0x000B	McBSP 接收通道使能寄存器——B 部分
XCERAx	0x000C	McBSP 传送通道使能寄存器——A 部分
XCERBx	0x000D	McBSP 传送通道使能寄存器——B 部分
PCRx	0x000E	McBSP 引脚控制寄存器

下面以 McBSP0 为例，说明 McBSP 串口控制寄存器（SPCR10 和 SPCR20）的配置方法。

```
SPSA0       .set   38h                    ; 定义子地址寄存器映射位置
SPSD0       .set   39h                    ; 定义子块数据寄存器映射位置
SPCR10      .set   00h                    ; 定义 SPCR10 的映射子地址
SPCR20      .set   01h                    ; 定义 SPCR20 的映射子地址
STM     SPCR10 ,SPSA0                     ; 将 SPCR10 的地址写入 SPSA0
STM     #K_SPCR10_CONFIG,SPSD0            ; 将配置值写入 SPSD0
STM     SPCR20 , SPSA0                    ; 将 SPCR2 的地址写入 SPSA0
STM     #K_SPCR20_CONFIG，SPSD0           ; 将配置值写入 SPSD0
```

在将 SPCR10 和 SPCR20 的子地址写入 SPSA0 之后，再将数据写入 SPSD0，内部复接器就自动将配置数值 K_SPCR10_CONFIG 和 K_SPCR20_CONFIG 分别写入子地址寄存器 SPCR10 和 SPCR20 中，从而完成寄存器配置。

McBSP 的状态主要由 SPCR1x/2x 中的比特控制（如表 9-18 所示）。例如当 SPCR1x 中的 RRST 设置为 1，McBSP 的接收器处于复位状态；而当 SPCR2x 中的 XRST 设置为 1，McBSP 的发送器处于复位状态。由于读取 DRR[1,2]或向 DXR[1,2]写数据时会影响 RRDY 和 XRDY 比特位，因此可以用来判定 McBSP 接收器和发送器的就绪状态。此外，利用 FRST 和 GRST 还可以控制帧同步产生器和采样率产生器的复位。

表 9-18　SPCR1x/2x 的比特定义

SPCR1x			SPCR2x		
位置	比特域	功能说明	位置	比特域	功能说明
15	DLB	数字环回模式	15~10	Reserved	保留
14~13	RJUST	接收符号扩展调整模式			
12~11	CLKSTP	时钟停止模式	9	FREE	自由运行模式
10~8	Reserved	保留	8	SOFT	软控制模式
7	DXENA	DX 延迟使能	7	FRST	帧同步产生器复位
6	ABIS	A-bis 模式	6	GRST	采样率产生器复位
5~4	RINTM	接收中断模式	5~4	XINTM	发送中断模式
3	RSYNCERR	接收同步错误指示	3	XSYNCERR	发送同步错误指示
2	RFULL	接收移位寄存器满指示	2	XEMPTY	发送移位寄存器满指示
1	RRDY	接收器准备	1	XRDY	发送器准备
0	RRST	接收器复位	0	XRST	发送器复位

McBSP 允许对接收、发送的数据时钟和帧同步进行独立的配置，例如 FSR、FSX、CLKX 和 CLKR 的极性、数据帧中的相位个数、字个数、字的比特数、帧同步到第一个数据比特的延迟以及接收数据时符号扩展方式等。这些传输特性的配置都可以在 RCR1x/2x，XCR1x/2x

和 PCRx 中找到对应的比特域，RCR1x/2x 和 PCRx 的比特定义分别如表 9-19 和表 9-20 所示。XCR1x/2x 的比特定义和 RCR1x/2x 相同，不再列出。

表 9-19　RCR1x/2x 的比特定义

位置	比特域	功能说明	位置	比特域	功能说明
15	Reserved	保留	15	RPHASE	接收相位
14~8	RFRLEN1	接收帧长度 1	14~8	RFRLEN2	接收帧长度 2
7~5	RWDLEN1	接收字长度 1	7~5	RWDLEN2	接收字长度 2
4~0	Reserved	保留	4~3	RCOMPAND	接收扩展模式
			2	RFIG	接收帧忽略模式
			1~0	RDATDLY	接收数据延迟

利用 RPHASE 可以设置数据帧包含一个或两个相位，每个相位的数据帧传送长度和字长都可以单独配置。数据帧长度可以设置为 1~128 个字，每个字的长度可以为 8,12,16,20,24,32 比特。利用 RCOMPAND 可以配置数据接收的字格式，可以实现 8 位数据的 LSB 传输、μ 律压扩和 A 律压扩（如图 9-13 中的压扩单元所示）。

表 9-20　PCRx 的比特定义

位置	比特域	功能说明	位置	比特域	功能说明
15	Reserved	保留	7	Reserved	保留
14			6	CLKS STAT	通用 IO 时 CLKS 引脚状态
13	XIOEN	发送通用 IO 模式	5	DX STAT	通用 IO 时 DX 引脚状态
12	RIOEN	接收通用 IO 模式	4	DR STAT	通用 IO 时 DR 引脚状态
11	FSXM	发送帧同步模式	3	FSXP	发送帧同步极性
10	FSRM	接收帧同步模式	2	FSRP	接收帧同步极性
9	CLKXM	发送时钟模式	1	CLKXP	发送时钟极性
8	CLKRM	接收时钟模式	0	CLKRP	接收时钟极性

当利用 McBSP 进行多通道传输时，可以对 MCR1x/2x、RECRAx，RECRBx 以及 XECRAx，XECRBx 等寄存器进行设置，实现所需要的功能（包括通道的使能、各通道长度的设置等）。限于篇幅，这里不再赘述，读者可以参考 TI 公司的技术文档《TMS320C54x DSP Reference Set Volume 5: Enhanced Peripherals》。

串口通信中比特率的设定非常重要。为了方便比特率的设定，DSP 为 McBSP 提供了一个单独的采样率发生器，它由三级时钟分频组成，如图 9-15 所示。采样率发生器寄存器 SRGR1/2 控制着采样率发生器的各种操作，其具体的比特定义如表 9-21 所示。

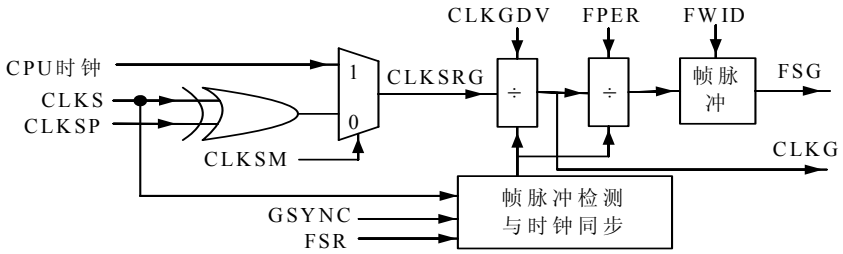


图 9-15　McBSP 时钟产生器示意图

采样率发生器可以为接口产生可编程的 CLKG（数据位时钟）信号和 FSG（帧同步时钟）

信号。CLKG 和 FSG 是 McBSP 的内部信号，用于驱动接收/发送时钟信号（CLKR/X）和帧同步信号（FSR/X）。采样率发生器时钟既可以由内部的 CPU 时钟驱动（CLKSM=1），也可以由外部时钟源驱动（CLKSM=0）。从图 9-15 中可以看出，采样率产生器产生的数据时钟 CLKG 与内部时钟源 CLKSRG 的分频比由 CLKGDV 确定，帧同步时钟 FSG 和 CLKG 的分频比由 FPER 确定，FWID 确定了帧同步信号的宽度。

表 9-21　采样率产生器寄存器 SRGR1/2 的比特定义

SRGR1			SRGR2		
位置	比特域	功能说明	位置	比特域	功能说明
15~8	FWID	帧同步信号宽度	15	GSYNC	SRGR 时钟同步
			14	CLKSP	时钟边沿极性选择
			13	CLKSM	SRGR 时钟模式
			12	FSGM	SRGR 发送帧同步模式
7~0	CLKGDV	采样率时钟分频比	11~0	FPER	帧分频比

9.5.2.3　McBSP的工作模式

利用丰富的寄存器控制功能，McBSP 可以实现不同的工作模式。除了利用正常连续工作模式与同步串口、PCM 接口、T1/E1 等多种接口相连外，McBSP 还可以工作在时钟停止模式、通用 I/O 模式。这些模式在实现简单串行接口、逻辑控制时非常方便。

1．简单串行接口（SPI）

SPI 协议是一种主从配置的、支持一个主方、一个或多个从方的串行通信协议，一般使用 4 条信号线：串行移位时钟线（SCK）、主机输入/从机输出线（MISO）、主机输出/从机输入线（MOSI）、低电平有效的使能信号线（SS）。SPI 接口的最大特点是由主设备时钟信号的出现与否来确定主/从设备间的通信。一旦检测到主设备时钟信号，数据开始传输，时钟信号无效后，传输结束，这期间需要从设备必须使能。

McBSP 具有时钟停止工作模式（由 CLKSTP 比特域控制），可以与 SPI 协议兼容。

SPI 协议区分主方和从方，因此 DSP 的 McBSP 与 SPI 器件相连时存在两种连接方案，如图 9-16 所示。其中图（a）和（b）分别是 MCBSP 作为主设备和从设备使用的情况。



（a）DSP 作为主设备　　（b）DSP 作为从设备

图 9-16　DSP 芯片利用 McBSP 引脚与 SPI 器件相连的示意图

McBSP 支持两种 SPI 传输格式，需要在 SPCR 寄存器的 CLKSTP 位中设置（当 CLKSTP=1x

时，时钟停止模式，10 时时钟没有延迟，11 时时钟有延迟）。具体的控制寄存器配置可按表 9-22 所示实现。

表 9-22　SPI 方式下 McBSP 的配置方法

比特域	所属寄存器	配置值	功能描述
CLKSTP	SPCR1	1xb	使能时钟停止模式，选择两种定时方式
CLKXP	PCR	0 或 1	配置 BCLKX 信号极性
CLKXM	PCR	0：输入 1：输出	配置 BCLKX 信号的方向 输入:从属设备　输出：主设备
RWDLEN1	RCR1	000-101b	接收数据包长度，必须等于 XWDLEN1
XWDLEN1	XCR1	000-101b	发送数据包长度，必须等于 RWDLEN1
CLKSM	SRGR2	1	直接由 CPU 时钟分频得到的采样率时钟
CLKGDV	SRGR1	1-255	定义采样率时钟分频比
FSXM	PCR	0：输入 1：输出	配置 BFSX 信号的方向 输入：从属设备　输出：主设备
FSGM	SRGR2	0	BFSX 信号由数据包传输触发
FSXP	PCR	1	配置 BFSX 低电平有效
XDATDLY	XCR2	00b 01b	为 BFSX 信号提供正确建立时间 从属设备必须为 0，主设备为 01b
RDATDLY	RCR2	00b 01b	为 BFSX 信号提供正确建立时间 从属设备必须为 0，主设备为 01b

2．通用I/O口

在两种情况下，串口的引脚（CLKX、FSX、DX、CLKR、FSR、DR 以及 CLKS）可以作为通用 I/O 口使用：

- McBSP 的某部分处于复位状态：此时 SPCR 寄存器中的（R/X）RST=0；
- 串口设置为通用 I/O 口：此时 PCR 中的（R/X）IOEN=1。

表 9-23 总结了这些引脚作为 I/O 口的用法。

表 9-23　McBSP 引脚作为通用 I/O 口使用的条件

引脚	通用 I/O 口条件	输出条件	输出值	输入条件	输入值
CLKX	$\overline{\text{XRST}}$ =0 XIOEN=1	CLKXM=1	CLKXP	CLKXM=0	CLKXP
FSX		FSXM=1	FSXP	FSXM=0	FSXP
DX		ALL	DX_STAT	NO	N/A
CLKR	$\overline{\text{RRST}}$ =0 RIOEN=1	CLKRM=1	CLKRP	CLKRM=0	CLKRP
FSR		FSRM=1	FSRP	FSRM=0	FSRP
DR		NO	N/A	ALL	DR_STAT
CLKS （C54x 没有）	$\overline{\text{RRST}}$ = $\overline{\text{XRST}}$ =0 RIOEN=1	NO	N/A	ALL	CLKS_STAT

9.5.2.4　McBSP的配置

McBSP 的编程配置主要分为初始化和对串行口缓冲器数据的访问控制两个部分。对串行口的初始化时，只需要向控制寄存器写入合适的配置值即可。具体步骤如下：

- 将 McBSP 发送器/接收器复位；
- 对和使用的接收器/发送器相关的寄存器进行编程；
- 唤醒接收器/发送器。

对相关寄存器编程时，需要设置如下控制位：全局控制、数据操作、帧同步和时钟产生。

- 全局控制：包括将接收器引脚设置为 McBSP 工作模式引脚、使能/禁止数字环回模式、使能时钟停止模式、使能/禁止接收的多通道选择模式等；
- 数据操作：包括设置接收字长和帧长、选择接收帧的相位数、使能/禁止接收帧同步忽略功能、设置接收器的扩展模式、设置接收数据延迟、设置接收符号扩展和调整模式、设置接收中断模式等；
- 帧同步和时钟产生：设置接收帧同步模式和极性、设置接收时钟模式和极性、设置 SRG 时钟分频模式、设置 SRG 时钟同步模式、配置帧同步周期和脉冲宽度、设置 SRG 时钟模式和输入时钟极性等。

在程序运行过程中访问串行口缓冲器的方法有两种：查询法和中断法。

（1）查询法：

查询串行口控制寄存器 SPCR1x/2x。通过判断 RRDY 和 XRDY 来确定串行口的状态，在数据接收缓冲器已满的情况下读取数据，在数据发送缓冲器已空的情况下写入数据。

（2）中断法：

此时需要将对应的 McBSP 中断使能，并启动串行口。要实现发送中断，需要先向 DXR 写入第一个数据值，之后数据的接收和传送可以在中断服务子程序（ISR）中完成。由于接收和传送是独立的，因此具有不同的中断矢量序号，接收和传送任务可以根据需要选择在一个 ISR 或不同的 ISR 中完成。

串行口的 ISR 需要完成的功能有：

（1）保存堆栈中任何被修改的内容；

（2）读 DRR 或写 DXR；

（3）恢复保存在（1）中的内容；

（4）返回 RETE。



9.5.3 主机接口（HPI）

在利用 DSP 芯片构建多机系统时，芯片之间需要共享数据。为此，DSP 芯片设计了主机接口（Host-Port Interface，HPI），用于外部处理器对 DSP 的存储器资源进行访问。此时，主机（也称为上位机）控制着该接口的主控权，通过它直接访问 DSP 的存储空间和外围设备。DSP 作为从机（也称为下位机），向主机开放总线的控制权。HPI 为 DSP 芯片的接口开发，尤其是多机协同工作提供了一种极为方便的途径。本节主要介绍 HPI 的配置和使用方法。

9.5.3.1 概述

主机口 HPI 是一个并行端口，根据数据总线位数的不同，HPI 可以按照 HPI-8 和 HPI-16 两种方式工作。C542、C545、C548 和 C549 是标准 8 位 HPI 接口，C5402 和 C5410 是增强型的 8 位 HPI 接口，C5416 和 C5420 为 16 位增强型 HPI 接口。

C54x DSP 片内的 HPI 接口主要由五个部分组成，其典型框图如图 9-17 所示。

（1）HPI 存储器（DARAM）：用于 TMS320C54x 与主机间传送数据；

（2）HPI 地址寄存器（HPIA）：由主机对其直接访问，存放当前寻址 HPI 存储单元的地址；

（3）HPI 数据锁存器（HPID）：由主机对其直接访问，存放当前进行读/写的数据；

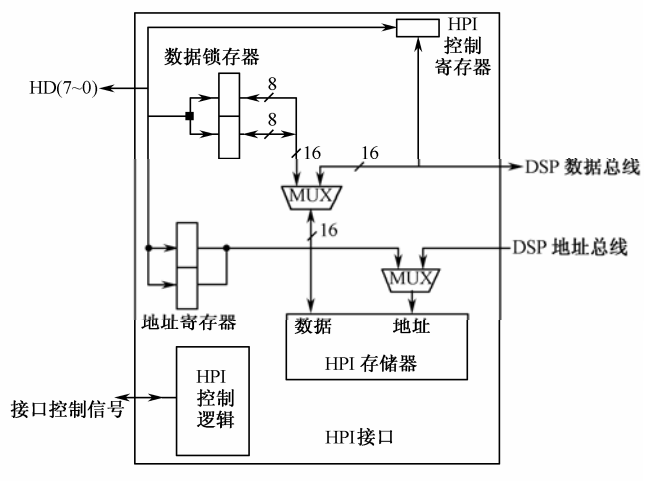


图 9-17　HPI-8 方框图

（4）HPI 控制寄存器（HPIC）：TMS320C54x 和主机都能对其直接访问，用于主处理器与 DSP 相互握手，实现相互中断请求；

（5）HPI 控制逻辑：用于处理 HPI 与主机之间的接口信号。

在这些部分共同作用下，HPI 具有如下主要特点：

（1）接口所需外围硬件很少；

（2）HPI 单元允许芯片直接利用一个或两个数据选通信号、一个独立或复用的地址总线、一个独立或复用的数据总线接到微控制单元 MCU 上；

（3）主机和 DSP 可独立地对 HPI 接口操作；

（4）主机和 DSP 握手可通过中断方式来完成。

另外，主机还可以通过 HPI 接口装载 DSP 应用程序、接收 DSP 运行结果或诊断 DSP 运行状态等。

9.5.3.2　HPI配置

HPI 作为从设备，为 DSP 和主机相连提供灵活而方便的接口。其接口外围电路比较简单，信号可分为以下几类：

- 数据总线： HD0~HD7（HPI-8）或 HD0~HD15(HPI-16)；
- 地址总线： HA0~HAn（不同的 DSP 芯片 n 不相同）；
- 控制线： $\overline{\text{HDS1}}$、$\overline{\text{HDS2}}$、$\overline{\text{HCS}}$、$\overline{\text{HAS}}$，HR/$\overline{\text{W}}$，HCNTL0，HCNTL1，HBIL；
- 握手线： HRDY，HINT 。

由于 HPI-8 和 HPI-16 存在一定的区别，下面分别进行介绍。

1．HPI-8

HPI-8 为 8 位并行口。16 位数据传输需要分为高低两个字节，用 HBIL 信号线来控制传输的是高字节还是低字节。主机和 HPI-8 的通信通过专用的地址和数据寄存器完成。

主机首先初始化 HPIC，然后设置 HPIA。当完成一个 HPIA 的写操作后，一个存储器的预取数开始执行。最后对 HIPD 进行操作，就可以对 TMS320C54x 的片内 RAM 进行读写。DSP

芯片不能直接访问这些专用的地址和数据寄存器，但可以对控制寄存器进行访问。

主机通过 HCNTL0 和 HCNTL1 两个引脚的配置来决定对 HPIC、HPID 和 HPIA 几个寄存器的访问，访问方法如表 9-24 所示。

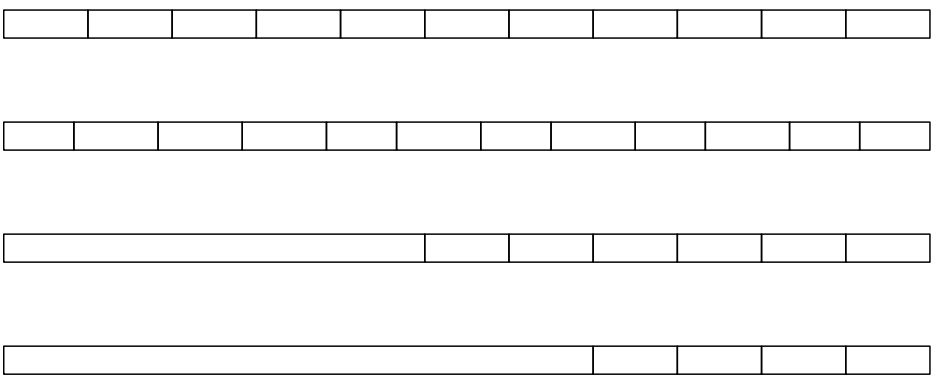
表 9-24　HCNTL0/1 模式

HCNTL1	HCNTL0	主机访问模式
0	0	主机能够读写 HPIC 寄存器
0	1	主机能够读写 HPID 寄存器，每访问一次 HPI 后，HPIA 寄存器值自动加 1
1	0	主机能够读写 HPIA 寄存器
1	1	主机能够读写 HPID 寄存器，HPIA 寄存器值不被修改

HPIC 寄存器（可寻址地址为 002CH）包括配置通信协议和控制通信（握手）的比特，如图 9-18 所示。各位的含义如下：

- BOB：字节选择位；
- DSPINT：主机向 TMS320C54x 发出中断位；
- HINT：TMS320C54x 向主机发出中断位；
- HPIENA：HPI 使能位，受引脚连接情况决定；
- XHPIA：扩展 HPI 地址，加*号表示只在 HPI 能够访问 DSP 扩展存储区间的芯片上存在。

当主机传送数据给 HPI 寄存器时，HPI 控制逻辑自动对内部 DARAM 中一个专用的 2 K 字的存储区进行访问，完成数据传送。C54x 可以通过访问自己的存储区来访问这些数据。



（a）主机读 HPIC；（b）主机写 HPIC；（c）TMS320C54x 读 HPIC；（d）TMS320C54x 写 HPIC

图 9-18　主机和 TMS320C54x 访问 HPIC 寄存器的结果

除了 HCNTL0/1 外，外部 HPI-8 接口控制信号还有$\overline{\text{HDS1}}$、$\overline{\text{HDS2}}$、$\overline{\text{HCS}}$、$\overline{\text{HAS}}$等，这为与不同主机相连提供了方便。不论主机具有一个或多个数据选通引脚、有没有地址锁存使能（ALE）信号，都可以和 HPI-8 相连。在这些控制信号中，$\overline{\text{HCS}}$作为基本使能输入引脚，$\overline{\text{HDS1}}$、$\overline{\text{HDS2}}$作为数据传输的控制引脚使用。在 HPI-8 内部，这些信号的连接如图 9-19 所示，因此如果需要，这些信号的逻辑可以互换使用。如果使用$\overline{\text{HCS}}$来控制 HPI-8 的访问周期，那么 HRDY 的操作将受其影响。由于两个数据选通信号是逻辑或的关系，因此将两者都置低并不产生一个使能条件。

15　13　12　11　10　9　8　7

X　XHPIA*　HINT　0　BOB

在 C5402 等型号的 DSP 芯片上具有增强型的 HPI-8 接口，其最大特点是允许主机访问 DSP 的整个片内空间，C5410 还能访问 DSP 存储器的扩展页。例如，C5410 的扩展存储器通过 23 位地址线寻址。增强型 HPI-8 首先将 HPIC 的 XHPIA 位置 1，然后将 23 位地址中的高 6 位写入 HPIA。接着，增强型 HPI-8 将 HPIC 的 XHPIA 位置 0，再将 23 位地址中的低 16 位写入 HPIA。和标准的 HPI-8 操作相似，每当完成一个 XHPIA 或 HPIA 的写操作后，一个存储器的预取数开始执行。

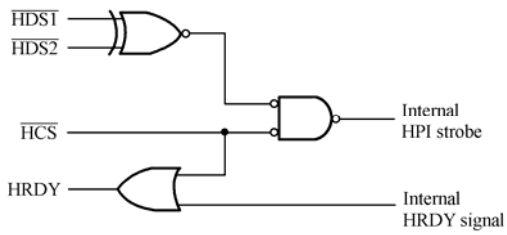


图 9-19　HPI 选通和选择逻辑示意图

DSP 的 HPI 与主机相连时，几乎不需要附加其他的逻辑电路。图 9-20 给出了 HPI-8 与主机连接的典型框图。

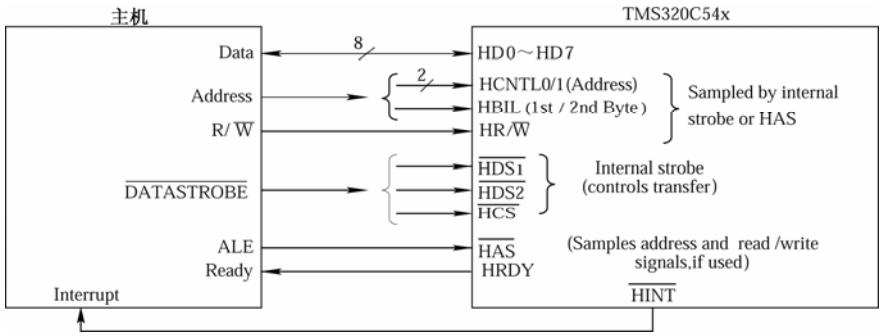


图 9-20　HPI-8 与主机连接的典型框

2．HPI-16

HPI-16 接口是在 HPI-8 基础上发展起来的增强 16 位数据接口。它提供了 16 比特并行双向数据总线，支持主处理器访问内部存储区域，无需将字拆分为字节。HPI-16 提供了多种外围设备与 TMS320 DSP 的接口。

HPI-16 主要特性包括：

- 16 位地址总线与主机地址总线直接连接（C5420 地址总线为 18 位）；
- 16 位数据总线（不需要字节排序）；
- 为多种 16 位主机提供了合适的选通和控制信号；
- 增加了复用与无复用操作方式，提高了接口灵活性；
- 存储区访问与 DMA 控制器同步，提供了在内部存储器地址范围内的访问；

HPI-16 提供了两种操作方式：复用方式和非复用方式。复用操作允许主处理器通过 HCNTL0 和 HCNTL1 控制 HPI 操作，而非复用方式只允许使用 HPI 地址和数据总线进行数据读/写。HPI 模式通过 HMODE 引脚进行硬件设置，复用模式 HMODE=0，非复用模式 HMODE=1。

（1）HPI-16 复用方式

在复用方式下，HCNTL0/1 控制 HPI 的操作。当主机和 DSP 之间需要使用中断，或者主

机的地址线和数据线需要复用时使用这种方式。主机通过两种方法控制 HCNTL0/1 和 HR/$\overline{W}$ 引脚。第一种情况是主机用指定的输出或地址线驱动 HCNTL0/1 和 HR/$\overline{W}$ 引脚。这种情况下，HCNTL0/1 和 HR/$\overline{W}$ 逻辑电平在主机$\overline{HDS}$信号下降沿被锁存。第二种情况是主机具有复用地址和数据总线。由于没有专门的地址总线，因此主机访问数据时使用同一总线驱动控制信号。这样，当允许主机随后执行写或读操作时，HPI 使用$\overline{HAS}$锁存 HCNTL0/1 和 HR/$\overline{W}$ 。

图 9-21 是一个使用$\overline{HAS}$信号的 C5420 芯片 HPI 复用方式的方框图。图中只有数据总线而没有地址总线。HCNTL 和 HR/$\overline{W}$ 引脚信号由主机驱动，地址锁存使能 ALE 与$\overline{HAS}$相连，用于锁存信号。

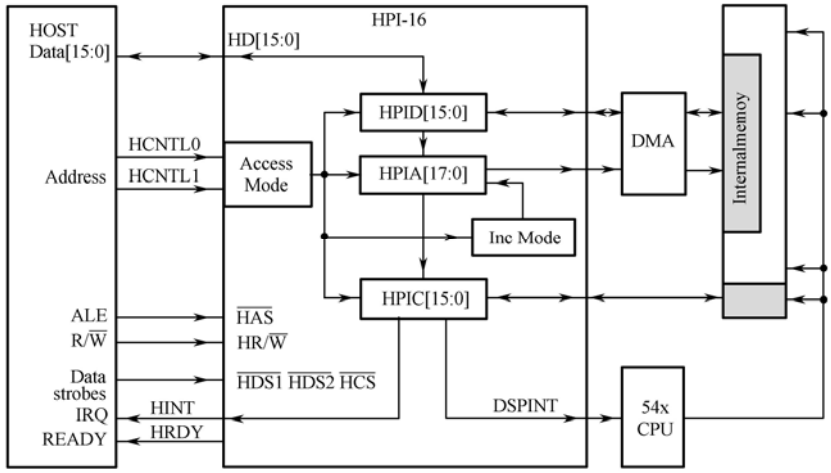


图 9-21　复用模式下 HPI-16 的接口示意图

在 HPI-16 中，HPIC 提供了更多的控制比特用于主机控制。其存储区域位于数据存储区地址 0x2C 处，在定义 HCNTL0/1=00b 时才能对该寄存器进行读写。图 9-22 和表 9-25 给出了 HPIC 的比特定义和描述。

15-6	5	4	3	2	1	0
RSVD	XHPIA	FETCH	HRDY	HINT	DSPINT	RSVD

图 9-22　增强型 HPI-16 接口中的 HPIC 寄存器

HPIA 寄存器包含了下一次数据访问的地址。在 HCNTL0/1＝10b 模式下，HPIA 地址自动增加。当 FETCH=1 时产生一个读请求，读数据由 HPIA 指向的地址来确定。FETCH 可以和 HRDY 比特一起用来软件查询，以确定内部访问完成的时间。当要更新 HPID 寄存器时，需要使用 FETCH 和 HRDY。执行一次 FETCH 操作不会增加 HPIA 寄存器的值。

主机和 DSP 的中断可以通过设置和查询 HPIC 的 DSPINT 及 HINT 位产生。主机设置 DSPINT=1，即发送一个中断给 DSP。如果 DSP 在 IMR 寄存器中设置了相应的中断允许位，那么 DSP 就能够产生该中断。同样地，DSP 能够通过向 HINT 写 1 产生到主机的中断。在这种情况下，主机必须也要向 HINT 写 1 来证实 HINT 请求。DSPINT 总是读为零，对 HINT 和 DSPINT 写零无效。

表 9-25　HPIC 位描述

比　特	访　问		描　述
	主　机	DSP	
DSPINT	R-0/W	R-0	主机通过对 DSPINT 写 1 来产生到 DSP 的中断；DSP 对该位进行写操作无效
HINT	R/W-1	R/W-1	DSP 到主机的中断；DSP 通过向 HINT 位写 1 来产生到主机的中断；主机向 HINT 写 1 清除 HINT 信号；主机或 DSP 对其写 0 无效
HRDY	R	R	HRDY 引脚的逻辑电平；主机和 DSP 均能读出该位，进行软件查询；如果 HRDY=0，则表明 HPI-16 没有完成当前的数据访问
FETCH	R-0/W	R-0	主机取数请求；当对该位写 1，则位于当前 HPIA 指定地址的数据被取出，存放在 HPID 寄存器中；主机和 DSP 读该位均为 0
XHPIA	R/W	R	扩展地址模式有效；当 XHPIA=1，主机对 HPIA 的写操作存放在 HPIA 的高位 HPIA[n:16]；如果 XHPIA=0，主机对 HPIA 的写操作存放在 HPIA[15:0]；在自增模式下，所有 n+1 位地址增加；读 HPIA 寄存器将按其同样方式执行操作；只有主机能够访问该位

由于数据总线为 16 位，但是地址可能超出 16 位（如‘VC5420 为 18 位），所以需要 XHPIA 比特来确定如何初始化 HPIA 地址位。当 XHPIA=0 时，装入 HPIA 的低 16 位；当 XHPIA=1 时，装入其扩展地址（大于 16 位）。建议在数据访问之前初始化所有的地址位。XHPIA 必须随 HPIA 寄存器的内容而变化。

（2）HPI-16 非复用方式

在非复用方式下，主机具有独立的地址线和数据线。主机使用 16 位的双向地址线可以访问 HPID 寄存器；使用 n 位的数据线可以访问 HPIA 寄存器。在非复用方式下，没有 HPIC 寄存器，因此主机和 DSP 之间没有中断（DSPINT 和 HINT）传递。由于有了主机的地址总线驱动 HPIA，因而不需要采用自增模式，也排除了使用 FETCH 和 XHPIA 比特的可能性。虽然不再使用软件对 HRDY 的查询，但是 HRDY 仍然是有作用的，它能够在内部访问 DSP 完成之前延迟随后的主机访问。HRDY 的功能在前面复用模式时已经做过说明。图 9-23 给出了非复用方式下与 HPI 的接口方法。

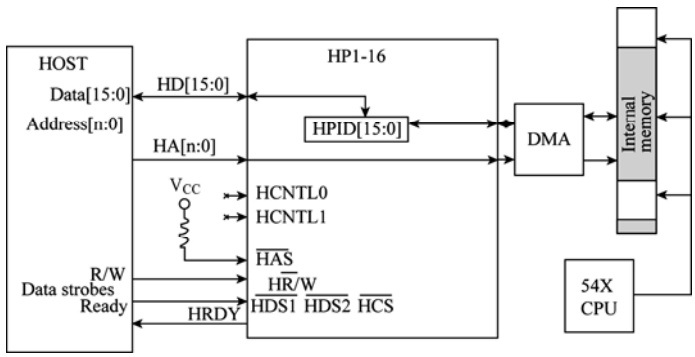


图 9-23　非复用模式下 HPI-16 的接口示意图

在非复用方式下，$\overline{\text{HAS}}$、HCNTL0 和 HCNTL1 引脚不再使用。当复用方式不使用$\overline{\text{HAS}}$或者自增操作时，非复用方式与复用方式的数据吞吐量相同。

9.5.4　外部存储器接口（EMIF）

EMIF（External Memory Interface）是外部存储器和 DSP 芯片内部其他单元间的接口。在

C55x 以及 C6000 等系列的 DSP 上，CPU 访问片外存储器必须通过 EMIF。EMIF 将多种不同的接口方式统一用相应的 DSP 芯片片内资源来实现，简化了 DSP 和外部存储器的接口，方便了系统设计。本节将以 C55x 为例具体介绍 EMIF 的控制及相关的设计。

9.5.4.1 概述

C55x 系列 DSP 的 EMIF 具有较强的接口能力，其数据总线宽度为 32bit，可寻址空间为 4GB，可以与基于并口的所有类型存储器直接相连。EMIF 的接口示意图如图 9-24 所示，可以看出它所支持的器件类型有如下几种：

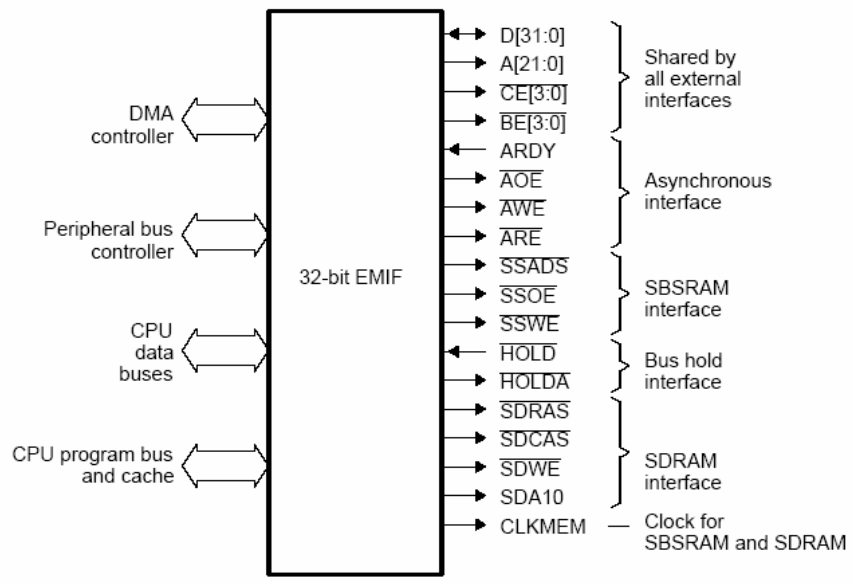


图 9-24　EMIF 的接口示意图

- 同步器件，包括同步突发 SRAM、同步 DRAM 等，速率均为 CPU 时钟速率的 1/2 或等于 CPU 时钟速率；
- 异步器件，包括异步 SRAM、ROM 和 Flash 存储器等。

EMIF 所处理的外部总线请求有四种来源：

- 片内程序存储控制器（PMC）发出的 CPU 程序取指请求；
- 片内数据存储控制器（DMC）发出的 CPU 数据存取请求；
- 片内 DMA 控制器；
- 外部共享存储器的设备。

外部存储空间	字地址	字节地址	大小
CE 0	02 8000~ 1F FFFF	05 0000~ 3F FFFF	2 M~160 K 字
CE 1	20 0000~ 3F FFFF	40 0000~ 7F FFFF	2 M 字
CE 2	40 0000~ 5F FFFF	80 0000~ BF FFFF	2 M 字
CE 3	60 0000~ 7F FFFF	C0 0000~ FF FFFF	2 M 字

图 9-25　EMIF 寻址空间分配示意图

当同时有多个部分请求外部总线时，由 EMIF 根据设置的优先级进行仲裁，然后响应各个申请。

EMIF 是并行总线，其引脚虽然多，但可以按表 9-26 分类。其寻址空间被划分为四个不同的空间，可以分别用于访问四个不同的芯片。每个空间可以访问的空间大小如图 9-25 所示。

表 9-26　EMIF 信号分类（n=0~3）

信号类别	信号名	信号方向	说　　明
数据总线信号	D[31:0]	I/O/Z	32 位 EMIF 数据线
地址总线信号	A[21:0]	O/Z	22 位 EMIF 地址线
片选信号	CEn	O/Z	芯片片选引脚，每个 CE 空间一个
字节使能信号	BEn	O/Z	字节使能引脚
总线保持控制信号	HOLD	I	保持请求引脚
	HOLDA	O	保持响应引脚
异步控制信号	ARDY	I	异步准备引脚
	AOE	O/Z	异步输出使能引脚
	AWE	O/Z	异步写选通引脚
	ARE	O/Z	异步读选通引脚
同步突发控制信号	SSADS	O/Z	SBSRAM 的地址选通、使能
	SSOE	O/Z	SBSRAM 的输出缓冲使能
	SSWE	O/Z	SBSRAM 的写使能
同步动态 RAM 控制信号	SDRAS	O/Z	SDRAM 的行选通信号
	SDCAS	O/Z	SDRAM 的列选通信号
	SDWE	O/Z	SDRAM 的写使能信号
	SDA10	O/Z	A10 地址线/自动预 charge 禁止
同步 RAM 时钟信号	CLKMEM	O/Z	存储器时钟引脚

9.5.4.2　EMIF配置

和 EMIF 相关的寄存器包括 EMIF 全局控制寄存器（EGCR）、EMIF CE0-CE3 空间控制寄存器，以及 EMIF SDRAM 时序控制寄存器等。其中 EGCR 可以用于配置和监视 EMIF 的全局状态，如监视 ARDY、HOLD 和 HOLDA 等引脚。EMIF 每个空间都有自己独立的控制寄存器，这样可以为 DSP 同时配置四个不同的外部存储空间，其比特定义如图 9-26 和表 9-27 所示。当选择为异步 RAM 时，需要对 CEn 的空间控制寄存器中其他的比特域（读/写建立、选通和保持等）进行初始化，而选择为同步 RAM 时，不需要对其中比特进行初始化。

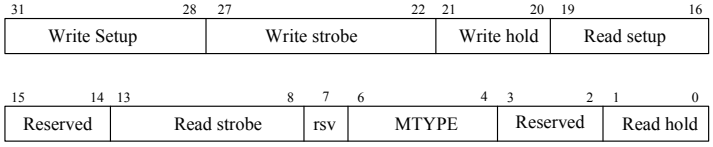


图 9-26　CEn 的空间控制寄存器

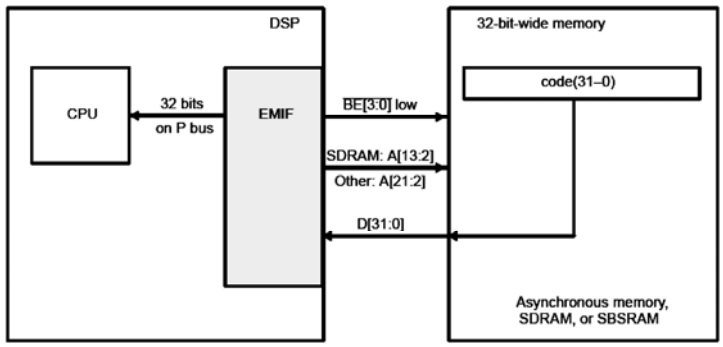
表 9-27　MTYPE 的可选类型

MTYPE	存储器类型
000b	8 位 ROM（CE1）
001b	16 位 ROM（CE1）
010b	32 位异步接口
011b	32 位 SDRAM（CE0，CE2，CE3）
100b	32 位 SBSRAM
其他	保留

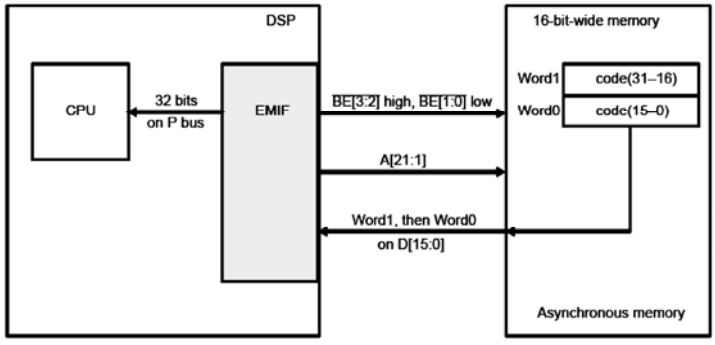
利用 EMIF 可以和程序存储器、数据存储器相连。由于 C55x DSP 芯片片内程序区的地址单元是按 8 位比特（字节）寻址的，数据区的地址单元是按 16 位比特宽度（字）寻址的，因

此在 EMIF 接口连线时需要注意地址线的变化。

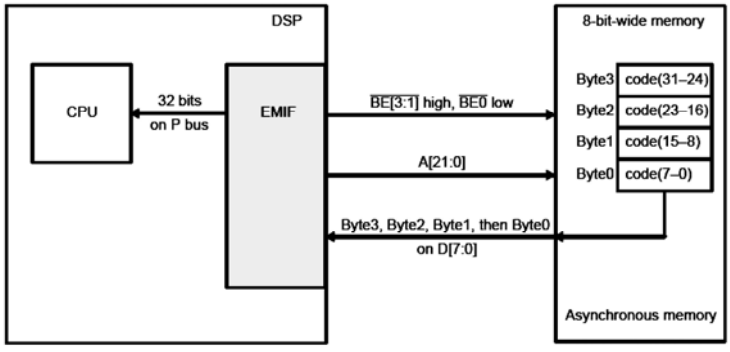
当从外部存储区读取指令代码时，EMIF 必须从外部空间读取 32 比特数据，并将它们置于 CPU 的程序读（program-read）数据总线上。在连接时，有三种存储器件宽度可以选择：32 比特、16 比特和 8 比特。图 9-27 是这三种不同的外部存储器件连接示意图。由于宽度不同，地址线需要和存储器件的基本寻址单元相对应，同时字节使能引脚 BE[3:0]也需要做相应调整。在后两种连接图中，DSP 分别需要读取 2 次 16 位数据和 4 次 8 位数据才能完成一个 32 位字的读取。每读一次，EMIF 接口的地址自动增 1。



（a） EMIF 与 32 位宽存储器的连接示意图



（b） EMIF 与 16 位宽存储器的连接示意图



（c） EMIF 与 8 位宽存储器的连接示意图

图 9-27　EMIF 接口连接示意图

在利用外部存储区对数据进行读写时，可以对 32 位、16 位和 8 位数据进行操作。可用于数据空间的存储器件宽度可以为 32 位和 16 位。对于 32 位数据读写操作，EMIF 必须和两个 16 位 CPU 总线进行通信。对于有些 CPU 指令和 DMA 控制器的操作来说，它们需要访问 8 位数据，这时，可以在 16 位宽和 32 位宽的存储器件中实现，然而读写过程存在着区别：当写字节时，EMIF 只改变一个字节内容，而读字节时，EMIF 读取的宽度等于存储器件的存储宽度，并在 DSP 内部将需要的字节提取出来。

对于不同类型的存储器件，EMIF 都可以方便的连接。图 9-28、图 9-29 和图 9-30 分别给出了 DSP 通过 EMIF 和 32 位异步存储器、32 位同步突发静态 RAM（SBSRAM）和 16 位同步动态 RAM（SDRAM）的连线示意图。由于提供了标准的引脚，EMIF 和这些器件的相连不需要扩展逻辑，在配置时也只需要将 MTYPE 设置为相应的类型，并对同步时钟、时序宽度进行合理配置即可。

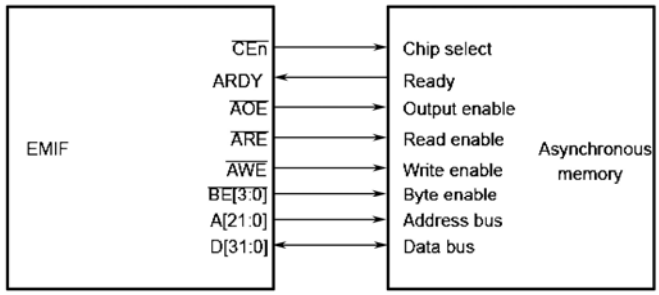


图 9-28　EMIF 和异步存储器的连线示意图

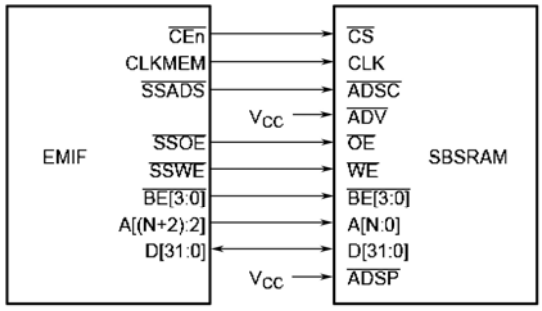


图 9-29　EMIF 和 SBSRAM 的连线示意图

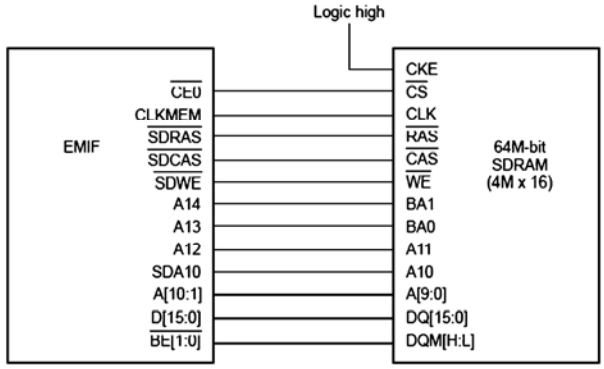


图 9-30　EMIF 和 SDRAM 的连线示意图

9.5.5 通用 I/O 口

为方便逻辑控制信号的输入/输出，一般的芯片上都提供有通用输入/输出（I/O）引脚。DSP 主要应用于大型的信号处理的计算任务中，但为了实现必要的逻辑控制，也设计了一定的通用 I/O 口，供开发者使用。除了 2.6 节中介绍的 XF 和$\overline{\text{BIO}}$两个通用 I/O 引脚外，一些 C54x DSP 芯片的多通道缓冲串口（McBSP）和主机接口（HPI）的一些引脚也可以作为通用 I/O 引脚。以 C5416 为例，可以作为通用 I/O 引脚的有：

（1）18 个 McBSP 引脚：当 MCBSP 的发送和接收部分处于复位状态时，可以将 BCLKX0/1/2，BCLKR0/1/2，BDR0/1/2，BFSX0/1/2，BFSR0/1/2，BDX0/1/2 作为通用 I/O 引脚。引脚主要由引脚控制寄存器（PCR）控制，具体细节可以参阅 TI 公司的技术文档《TMS320C54x DSP Reference Set Volume 5: Enhanced Peripherals》中对 McBSP 的相关介绍。

（2）8 个 HPI 引脚：当 HPI 以 HPI-16 方式（HPI16＝1）运行，或者 HPI 被屏蔽（HPIENA＝0）时，可以将 HPI 的 8 位的双向数据总线 HD0~HD7 作为通用 I/O 引脚。有两个存储器映射寄存器可用于控制 HPI 的通用 I/O 操作：通用 I/O 控制寄存器（GPIOCR）和通用 I/O 状态寄存器（GPIOSR）。

GPIOCR 的比特分配如图 9-31 所示，其中 DIRx 用于配置 HD0—HD7 为输入还是输出。

15	8	7	6	5	4	3	2	1	0
Reserved		DIR7	DIR6	DIR5	DIR4	DIR3	DIR2	DIR1	DIR0
0		R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

图 9-31　GPIOCR 的比特分配

通用 I/O 的引脚状态可以通过 GPIOSR 的比特位来监视，GPIOSR 的比特分配如图 9-32 所示，每个比特对应于一个通用 I/O 引脚。

15	8	7	6	5	4	3	2	1	0
Reserved		IO7	IO6	IO5	IO4	IO3	IO2	IO1	IO0
0		R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

图 9-32　GPIOSR 的比特分配

9.6　基于 CSL 的外设开发

由于 DSP 芯片片内集成外设种类及其应用日趋复杂，为了方便用户编写配置和控制它们所必需的定义和代码，DSP 芯片开发商通常会为用户提供一组标准的方法用于访问和控制片内集成外设。TI 公司在 CCS 开发环境中推出了一套软件库用于外设的开发，称为芯片支持库（Chip Support Library，CSL）。

由于各种系列的 DSP 片内集成外设实现有区别，因此 TI 公司分别为 C6000 和 C5000 系列 DSP 设计了各自的 CSL 库。

9.6.1　CSL 概述

CSL 库函数大多数是用 C 语言编写的，它通过定义一组标准的用户程序接口（APIs）：函数、数据类型、宏，实现了外设编程的标准协议，而且已对代码的大小和速度进行了优化。熟悉 C 语言编程的用户可以方便地在 CSL 库函数基础上对集成外设进行开发，而不需要再编写过多的汇编源程序控制外设，缩短了开发时间，增加了代码可移植性。

CSL 库中已经实现的模块有：

- CSL：顶层模块（top level module）；
- DAT：和器件无关的数据拷贝/填充模块（Device independence data copy/fill）；
- CHIP：和器件相关的定义模块（Device Special module）；
- DMA：直接存储器访问模块（Direct memory access）；
- EBUS：外部存储器总线接口模块（external memory bus interface）；
- GPIO：通用 I/O 模块（general purpose I/O）；
- HPI：主机接口模块（Host port interface）；
- IRQ：中断控制模块（Interrupt controller）；
- MCBSP：多通道缓冲串口模块（Multichannel buffered serial port）；
- PWR：功耗控制模块（Power down）；
- STDINC：标准包含模块（Standard include module）；
- TIMER：定时器模块（Timer module）。

除此之外，CSL 还具有如下一些良好的特点：

（1）CSL 库是可裁剪的：即只有被使用的 CSL 模块才会包含进应用程序中；

（2）CSL 库是可扩展的：每个集成外设的 API 相互独立，增加新的 API，对其他片内外设没有影响；

（3）CSL 库对硬件进行抽象，提取符号化的片内外设描述：大多数常用的寄存器及其域值都已经通过宏进行了定义，因此可以直接利用宏进行访问和建立。

9.6.2　CSL 程序开发

由于 CSL 库已经将各种芯片外设控制功能利用 C 语言实现，因而利用 CSL 编写程序时，只需要调用相应的 API 函数即可。

CSL 已经集成在 CCS2.0 及以上版本的 BIOS 配置工具中，因而有两种利用 CSL 进行配置的方法：

（1）通过定义和初始化 objects 来配置外设，并在自己的 C 源代码中调用 CSL 函数；

（2）利用 CSL 的图形用户界面（GUI）来配置外设。

用户可以根据自己的需要采用函数代码或图形方式开发自己的片内外设应用程序，下面分别予以介绍。

9.6.2.1　利用CSL图形用户界面使用CSL

CSL 图形用户界面（GUI）集成在 DSP/BIOS 配置工具树中，可以方便用户对一些片内外设进行配置。

用户可以在 CCS 中选择 file→new→DSP/BIOS Configuration 生成自己的配置文件，默认名为 config1.cdb。每个外设都表示为图 9-33 中 CSL 树的一个子目录。

每个外设都由两个部分组织而成，一个是外设配置管理器（Configuration Manager），另一个是外设资源管理器（Resource Manager）。其中，配置管理器可以用于建立和配置外设对象。如图 9-34 所示，在 Configuration Manager 图标上单击鼠标右键，可以在弹出菜单中选择新建外设对象。对于已有的对象，也可以对相应的外设寄存器进行配置（如图 9-35 所示为 DMA 对象）。每个外设有其自己的控制寄存器，用户可以根据自己的需要逐比特域的进行配置（如图 9-36 所示为 DMA 对象的属性管理界面）。

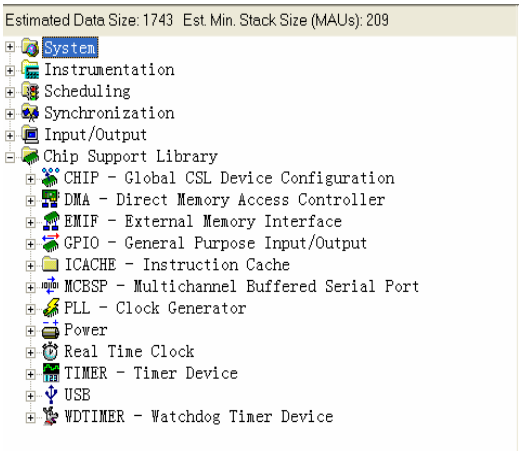


图 9-33　CSL 树界面示意图

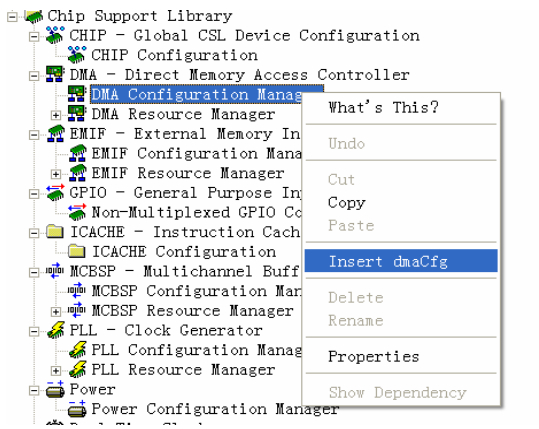


图 9-34　新建外设对象示意图

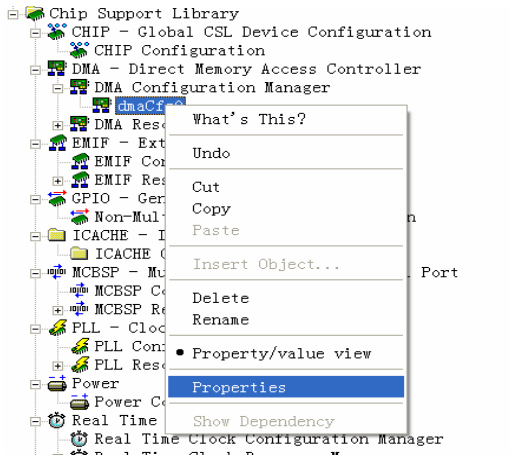


图 9-35　外设对象属性配置操作示意图

配置完所有的外设对象后，将该 cdb 文件加到自己的项目中。这时就可以利用这些定义的外设对象了。

【例 9.7】以定时器配置为例进行 CSL GUI 的使用。

在已按正常步骤生成 mytimer.cdb 配置文件的前提下，可按如下步骤进行：

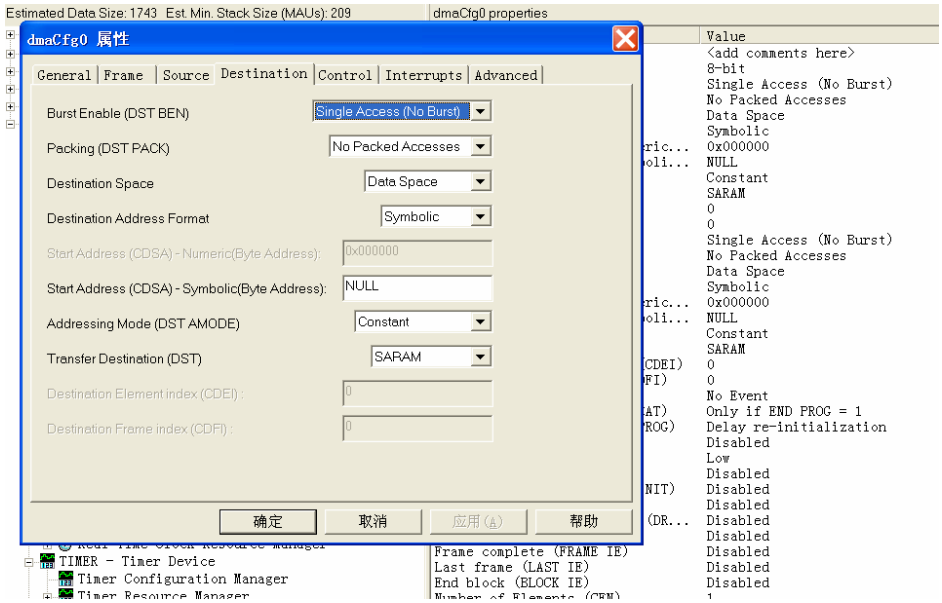


图 9-36　DMA 对象的属性管理界面

（1）在 TIMER 配置管理器中插入一个新的配置对象。

（2）右键单击 timerCfg0，选择 Properties 并配置 timerCfg0（如图 9-37 所示）。

（3）在 Timer 资源管理器下，右键单击 TIMER1，并选择 Properties 打开 TIMER1 属性窗口（如图 9-38 所示）。为 TIMER1 选择句柄（Handle）hTIMER1，并使能预初始化，在预初始化的下拉列表中，选择配置对象 timerCfg0 作为 TIMER1 的初始配置。



图 9-37　timer 对象的初始化配置

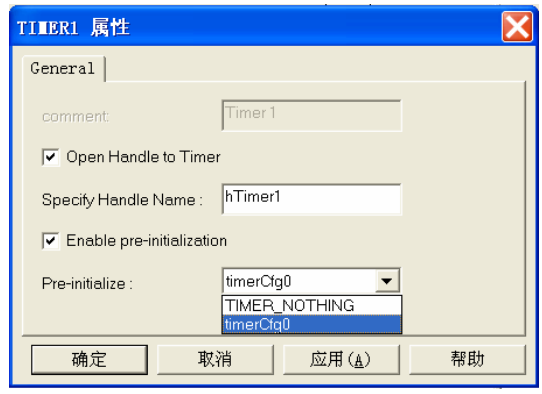


图 9-38　timer 对象的句柄和初始化使能

（4）将 mytimer.cdb 保存后，会自动生成一系列文件：

- mytimer.cdb：包含了所有的对象和配置；
- mytimercfg.h：对所有新建对象的外部声明，包括芯片类型的定义和 CSL 模块头文件；
- mytimercfg_c.c：新建对象的定义和初始化，包含了用于预开启和预初始化对象的 TIMER_open（打开函数）和 TIMER_config（配置函数）；
- mytimercfg.sXX：定义了存在的 DSP/BIOS 元素，并提供了这些元素的初始化；
- mytimercfg.cmd：连接命令文件，包含有 CSL 库和.csldata 存放位置。

其中 mytimercfg.h 和 mytimercfg_c.c 是专用于 CSL 的，并且使用了 CSL 的 API 函数。项目的源文件必须包含 mytimercfg.h。

下面是生成的头文件的源代码：

```
头文件   mytimecfg.h
#define CHIP_5510 1          //定义器件符号
#include <std.h >
#include <hst.h >
#include <swi.h>
#include <tsk.h>
#include <log.h>
#include <sts.h>
#include <csl_timer.h>       // 与 CSL 树中实现的外设相对应的 CSL 头文件

下面是一个主文件 main.c 的例子。
#include <mytimercfg.h >
Static Uint32 TIMEREventId1 ;    // 无符号 32 位整型数
void main()
{
      TIMEREventId1 = TIMER_getEventId(hTimer1) ; // 获取 TIMER 对象的事件 ID
      IRQ_enable(TIMEREventId1) ;                 // 使能 TIMER 事件
      TIMER_start(hTimer1) ;                      // 开启 TIMER
      While ( !IRQ_test(TIMEREventId1)) ;         // 等待 TIMER 中断
      TIMER_close(hTimer1) ;                      // 关闭 TIMER
}
```

函数中使用 TIMER_getEventId，TIMER_start，TIMER_close 都是定时器模块的 API 函数，IRQ_enable，IRQ_test 是中断控制模块的 API 函数。可以看到，利用这样的函数编写程序，程序的可读性更强，用户也不必再关注于各外设寄存器的比特域位置和具体的配置值，方便程序的开发。

9.6.2.2 不利用CSL图形用户界面使用CSL

有些开发者习惯于利用 C 语言手工定义和初始化外设配置对象。这样就不需要使用 GUI 了。但如果需要涉及很多中断管理，以及软中断、任务等调度管理功能，使用 GUI 更为方便。

使用 CSL 对外设进行编程有两种方法：

（1）基于寄存器的配置（PER_config()）：通过设置存储器映射寄存器的值来配置外设，直接利用寄存器进行配置，其代码大小和运行周期更为优化；

（2）基于功能参数的配置（PER_setup()）：通过一组功能参数来配置外设，和基于寄存器的配置方式相比，其代码大小和运行周期没有优势，但更为精练。

【例 9.8】利用 CSL 编写程序实现 DMA 通道 0 的数据传输。

使用基于寄存器的配置方法对 DMA 通道 0 进行初始化，将 0x3000 地址处的表格搬移到 0x20000 处。

基本的步骤如下：

- 在源文件中包含 csl.h 和 csl_dma.h；

- 定义和初始化 DMA 通道配置结构；
- 定义 DMA 句柄指针；
- 初始化 CSL 库；
- 对于多资源外设（如 McBSP 和 DMA），调用 PER_open 来使用这些资源，这里为 DMA_open 函数；
- 调用 DMA_config 来配置 DMA 通道资源；
- 调用 DMA_start 开始 DMA 传输；
- 当 DMA 传输完毕后关闭 DMA。

具体的代码如下所示。

```
#include <csl.h>
#include <csl_dma.h>

#define N 16              // 传输块大小
#pragma DATA_SECTION(src," table1")
Uint16 src[N]={0xBEEFu,0xBEEFu,……，0xBEEFu};     // 源数据表地址
#pragma DATA_SECTION(dst, "table2")
Uint16 dst[N];                    // 目的数据表地址
DMA_config myconfig = {
DMA_DMACSDP_RMK(0,0,0,0,0,0,1),          /* DMACSDP    */
DMA_DMACCR_RMK(1,1,0,0,0,0,0,0,0),       /* DMACCR     */
DMA_DMACICR_RMK(1,1,1,1,1,1),            /* DMACICR    */
(DMA_AdrPtr) &src,                       /* DMACSSAL   */
0,                                       /* DMACSSAU   */
(DMA_AdrPtr) &dst,                       /* DMACDSAL   */
0,                                       /* DMACDSAU   */
N,                                       /* DMACEN     */
1,                                       /* DMACFN     */
0,                                       /* DMACFI     */
0};                                      /* DMACEI     */
DMA_Handle myhDma;
void main(void)
{
    CSL_init();       // 初始化 CSL
    myhDma = DMA_open(DMA_CHA0,0);          /* 开启 DMA 通道 0 */
    myconfig.dmacssal = (DMA_AdrPtr)(((Uint16)(myconfig.dmacssal)<<1)&0xFFFF);
    myconfig.dmacdsal = (DMA_AdrPtr)(((Uint16)(myconfig.dmacdsal)<<1)&0xFFFF);
    DMA_config(mydHma, &myConfig);          /* 配置通道 */
    DMA_start(myhDma);                      /* 开始传输 */
    DMA_close(myhDma);                      /* 关闭通道 */
}
```

9.7 小　　结

本章重点讨论了有关 DSP 芯片的片内外设开发。由于 DSP 芯片种类越来越多，芯片外设的种类也很多，因此书中不能完全罗列。但芯片的外设开发是 DSP 系统设计和开发中的一个难点，需要读者重点掌握。在实际使用中，读者可以进一步参阅其他参考资料，了解相关内容。

9.8 习题与思考题

（1）C54x DSP 芯片中有哪些主要的片内外设？
（2）CPU 与外设传输数据的主要方式有哪些？试比较它们的优缺点。
（3）DSP 的中断处理步骤有哪些？
（4）DSP 中断矢量表的重定位如何实现？
（5）DSP 的 DMA 传输有哪几种传输方式？
（6）在时序模块的设计中，需要考虑哪些因素？
（7）McBSP 接口的工作模式有哪些？如何配置？
（8）熟悉 CSL 的 GUI 开发过程，编写与例 9.8 功能相同的 CSL GUI 程序。

第 10 章　DSP 脱机系统设计

10.1　引　　言

在 DSP 系统开发的初期阶段，软件部分的开发一般采用联机调试的方式进行，即通过运行在计算机上的软件开发环境来完成。开发设计人员可以通过软件开发环境对 DSP 系统进行代码加载、调试、运行等。在调试完成后，需要将软件固化到 DSP 系统，以便系统可以脱离计算机独立运行工作，通常我们将这种系统称为脱机系统。

本章以 TI 公司的 TMS320C54x 系列 DSP 芯片为主进行介绍，首先介绍 DSP 芯片的初始化，随后介绍 DSP 芯片的脱机系统设计。

10.2　DSP 芯片的初始化

DSP 芯片的初始化是设定 DSP 芯片工作状态的重要步骤，只有正确进行 DSP 芯片的初始化，才能保证芯片的正确运行。在软件调试过程中，如果发现程序运行不正确，应首先查看芯片各寄存器的初始化状态设置是否正确，然后再检查调试用户程序，以提高程序调试效率。本节以 TMS320C54x 为例，介绍 DSP 芯片的初始化方法。

10.2.1　寄存器初始化

TMS320C54x DSP 芯片加电复位之后处于预先设定的状态，无论是状态寄存器还是控制寄存器都有一个确定的数值。例如，DSP 芯片复位之后，控制寄存器 PMST 中的 IPTR 均为 1，从而将矢量映射到程序存储空间的第 511 页，这意味着复位矢量总是驻留在程序空间的 FF80H 处。此外，还需要对芯片的堆栈指针（SP）、存储器映射的外设控制寄存器（SWWSR 和 BSCR）、状态寄存器（ST0 和 ST1）和模式控制寄存器（PMST）等进行初始化，以适应不同应用的需求。

下面，我们给出一个对 TMS320C54x 的寄存器进行初始化的实例，这些寄存器包括：状态寄存器 ST0、状态寄存器 ST1、处理器模式控制寄存器 PMST、软件等待状态寄存器 SWWSR、组交换控制寄存器 BSCR 和时钟模式寄存器 CLKMD。在这个例子中，为便于读者理解，先给出各寄存器各比特域的定义和位置，然后给出各比特域需要初始化的值，分号后给出的是该比特域的复位值。

【例 10.1】 TMS320C54x 寄存器初始化

```
*  本程序对 ST0,ST1,PMST,SWWSR,BRCR,CLKMD 等寄存器进行初始化
;  （1）ST0 寄存器内部组织
*      ---------------------------------------------------------------
*      |15       13| 12 |    11 | 10    |     9    | 8                0 |
*      ---------------------------------------------------------------
```

```
*     |   ARP   | TC | C   | OVA | OVB |       DP          |
*     ------------------------------------------------------------
K_ARP          .set      000b<<13          ;ARP=111b at reset
K_TC           .set      1b<<12            ;TC = 1 at reset
K_C            .set      1b<<11            ;C = 1 at reset
K_OVA          .set      1b<<10            ;OVA = 0 at reset
K_OVB          .set      1b<<9             ;OVB = 0 at reset
K_DP           .set      00000000b<<0      ;DP=0 at reset
K_ST0          .set      K_ARP|K_TC|K_C|K_OVA|K_OVB|K_DP
;（2）ST1 寄存器内部组织
*     ------------------------------------------------------------------------------------
*     |  15  |  14 | 13  | 12 |  11  | 10  |  9 |  8  | 7  |  6  |  5  | 4    0 |
*     ------------------------------------------------------------------------------------
*     | BRAF | CPL | XF | HM | INTM | 0   |OVM|SXM|C16|FRCT|CMPT| ASM |
*     ------------------------------------------------------------------------------------
K_BRAF         .set      0b<<15            ;BRAF = 0 at reset
K_CPL          .set      0b<<14            ;CPL = 0    at reset
K_XF           .set      1b<<13            ;XF = 1     at reset
K_HM           .set      0b<<12            ;HM = 0    at reset
K_INTM         .set      1b<<11            ;INTM
K_ST1_RESR     .set      0b<<10
K_OVM          .set      1b<<9             ;OVM = 0    at reset
K_SXM          .set      1b<<8             ;SXM =1    at reset
K_C16          .set      0b<<7             ;C16 = 0    at reset
K_FRCT         .set      0b<<6             ;FRCT = 0 at reset
K_CMPT         .set      0b<<5             ;CMPT = 0 at reset
K_ASM          .set      00000b<<0         ;ASM = 0    at reset
K_ST1_HI       .set      K_BRAF|K_CPL|K_XF|K_HM|K_INTM|K_ST1_RESR
K_ST1_LO       .set      K_OVM|K_SXM|K_C16|K_FRCT|K_CMPT|K_ASM
K_ST1          .set      K_ST1_HI|K_ST1_LO
;（3）PMST 寄存器内部组织
*     ------------------------------------------------------------------------------------
*     | 15      7 |     6    |   5    |   4    |   3     |   2       |   1    |   0   |
*     ------------------------------------------------------------------------------------
*     |    IPTR    | MP/MC- | OVLY | AVIS | DROM | CLKOFF | SMUL | SST |
*     ------------------------------------------------------------------------------------
K_IPTR         .set      011111110b << 07        ;111111111b at reset
K_MP_MC        .set      1b << 6                 ;1 at reset
K_OVLY         .set      1b << 5                 ;0 at reset
K_AVIS         .set      0b << 4                 ;0 at reset
K_DROM         .set      0b << 3                 ;0 at reset
K_CLKOFF       .set      0b << 2                 ;0 at reset
K_SMUL         .set      0b<<1                   ;0 at reset
K_SST          .set      0b                      ;0 at reset
K_PMST         .set      K_IPTR|K_OVLY|K_DROM|K_CLKOFF|K_SMUL|K_SST
*（4）SWWSR 寄存器内部组织
*     ------------------------------------------------------------------------------------
```

```
*     |     15   | 14   12| 11       9| 8          6 |   5          3| 2                0|
*     ---------------------------------------------------------------------------------------
*     |   XPA   |   I/O   |   DATA   |   DATA   |   PROGRAM |   PROGRAM      |
*     ---------------------------------------------------------------------------------------
K_XPA               .set      0b << 15          ;0 at reset
K_IO                .set      000b << 12        ;111b at reset
K_DATA1             .set      000b << 9         ;111b at reset
K_DATA2             .set      000b << 6         ;111b at reset
K_PROGRAM1          .set      000b << 3         ;111b at reset
K_PROGRAM2          .set      000b              ;111b at reset
K_SWWSR             .set       K_XPA|K_IO|K_DATA1|K_DATA2|K_PROGRAM1|K_PROGRAM2
*（5）BSCR 寄存器内部组织
*       ---------------------------------------------------------
*       | 15       12|    11   | 10      3 |   2   |   1   |   0   |
*       ---------------------------------------------------------
*       | BNKCMP | PS-DS | Reserved | HBH | BH   | EXIO|
*       ---------------------------------------------------------
K_BNKCMP            .set      0000b << 12       ;1111b at reset
K_PSDS              .set      0b << 11          ;1 at reset
K_BSCR_RESV         .set      00000000b << 3
K_HBH               .set      0b<<2             ;0 at reset
K_BH                .set      0b<< 1            ;0 at reset
K_EXIO              .set      0b                ;0 at reset
K_BSCR              .set      K_BNKCMP|K_PSDS|K_BSCR_RESV|K_HBH|K_BH|K_EXIO
*（6）CLKMD 寄存器内部组织
*       ---------------------------------------------------------------------------------------
*       | 15       12|    11    | 10           3 |      2       |      1      |         0       |
*       ---------------------------------------------------------------------------------------
*       | PLLMUL | PLLDIV | PLLCOUNT | PLLON/OF | PLLNDIV| PLLSTATUS |
*       ---------------------------------------------------------------------------------------
K_PLLMUL            .set      0011b<<12
K_PLLDIV            .set      0b<< 11
K_PLLCOUNT          .set      00000000b<< 3
K_PLLON_OF          .set      1b<< 2
K_PLLNDIV           .set      1b<< 1
K_PLLSTATUS         .set      1b
K_CLKMD             .set
      K_PLLMUL|K_PLLDIV|K_PLLCOUNT|K_PLLON_OF|K_PLLNDIV|K_PLLSTATUS
_init_DSPC54x:
          SSBX          INTM                ;关闭中断总开关
          STM           #K_SWWSR ,SWWSR     ;初始化 SWWSR
          STM           #K_BSCR ,BSCR       ;初始化 BSCR
          STM           #K_ST0,ST0          ;初始化 ST0
          STM           #K_ST1,ST1          ;初始化 ST1
          STM           #K_CLKMD,CLKMD      ;初始化 CLKMD
          STM           #K_PMST,PMST        ;初始化 PMST
          STM           #0,IFR              ;中断标志寄存器 IFR=0
```

```
STM     #0,IMR          ;中断屏蔽寄存器 IMR=0
ORM     #1b<<6,*（IMR）  ;使能 brint2 中断
ORM     #1b,*（IMR）     ;使能 int0 中断
ORM     #1b<<1,*（IMR）  ;使能 int1 中断
RET
```

例中各寄存器比特域的初始化值由.set 后面的值确定。例如，下面语句将 CLKMD 寄存器中的 PLLMUL 初始化为 3：

```
K_PLLMUL    .set    0011b<<12
```

如果要修改相应的值，只需更改.set 后的值即可。如要设置 PLLMUL=9，则：

```
K_PLLMUL    .set    1001b<<12
```



10.2.2 中断矢量表初始化

中断矢量表是每个 DSP 系统必须用到的，对中断矢量表进行初始化是 DSP 初始化的一个重要组成部分。正确设置中断矢量包括：

（1）根据 DSP 芯片对各中断矢量的设置位置编写一个子程序，需要注意的是，在 TMS320C54x 中，要求每个中断矢量占 4 个字，如跳转指令不够 4 个字，可用 NOP 指令填充，每个 NOP 占一个字。

（2）设置 PMST 控制寄存器，该寄存器的高 9 位是 IPTR，用于设置矢量表的起始地址，如：IPTR=011111110b，则矢量表的起始地址为 7F00H。

（3）链接时将矢量表重定位到 IPTR 指定的地址。

下面是 TMS320VC5409 矢量表的一个例子，在这个例子中，_c_int00 是用户程序的最初入口点（由 C 编译器生成），_start 是用户程序入口点，ad_routine 是串行口 2 接收中断程序的入口点。

【例 10.2】 TMS320VC5409 矢量表初始化

```
.mmregs
.global   _c_int00
.global   ad_routine

.sect  "vectors"
reset: B     _c_int00          ; 复位矢量，该指令占两个字
NOP
NOP
nmi: RETE                      ; 非屏蔽中断，该指令占一个字
NOP
NOP
NOP
; 软中断
sint17      .space      4*16
sint18      .space      4*16
sint19      .space      4*16
sint20      .space      4*16
sint21      .space      4*16
sint22      .space      4*16
```

```
sint23      .space      4*16
sint24      .space      4*16
sint25      .space      4*16
sint26      .space      4*16
sint2       .space      4*16
sint28      .space      4*16
sint29      .space      4*16
sint30      .space      4*16
int0: RETE                  ; INT0 中断
NOP
NOP
NOP
int1: RETE                  ; INT1 中断
NOP
NOP
NOP
int2:  RETE
NOP
NOP
NOP
tint:   RETE                ; 定时器中断
NOP
NOP
NOP
brint0:RETE                 ; McBSP 串行口接收中断 0
NOP
NOP
NOP
bxint0: RETE                ; McBSP 串行口发送中断 0
NOP
NOP
NOP
brint2:BD       ad_routine ; McBSP 串行口接收中断 2，该指令占 2 个字
PSHM      ST0
PSHM      ST1
bxint2:RETE                 ; McBSP 串行口发送中断 2
NOP
NOP
NOP
int3:  RETE                 ; INT3 中断
NOP
NOP
NOP
hpint: RETE
       NOP
       NOP
       NOP
```

```
brint1:B        _start      ; McBSP 串行口接收中断 1，该指令占两个字
        NOP
        NOP
bxint1:B        _start      ; McBSP 串行口接收中断 1，该指令占两个字
        NOP
        NOP
.end
```



10.2.3 串行口初始化

串行口是 DSP 芯片上一种十分重要的外设接口。C54x 上集成了多通道缓冲串行口（McBSP），第 9 章已对此作了介绍。下面举例说明 McBSP 串行口的配置方法。

例 10.3 是对 TMS320C54x 的 McBSP2 串行口进行初始化配置的一个例子。在这个例子中，将串行口配置为 8 位字长、内部提供接口时钟、无压缩扩张。

【例 10.3】 McBSP2 串行口的初始化配置

```
---------- McBSP2 寄存器地址----------
DRR12           .set  0031H     ; 串行口 2 数据接收寄存器
DXR12           .set  0033H     ; 串行口 2 数据发送寄存器
SPSA2           .set  0034H     ; 串行口 2 Sub-bank 地址寄存器
SPSD2           .set  0035H     ; 串行口 2 Sub-bank 数据寄存器
---------- McBSP 控制寄存器子地址----------
SPCR1_SUBADDR   .set  0000H     ; 串行口控制寄存器 1（子地址）
SPCR2_SUBADDR   .set  0001H     ; 串行口控制寄存器 2（子地址）
RCR1_SUBADDR    .set  0002H     ; 接收控制寄存器 1（子地址）
RCR2_SUBADDR    .set  0003H     ; 接收控制寄存器 2 （子地址）
XCR1_SUBADDR    .set  0004H     ; 发送控制寄存器 1 （子地址）
XCR2_SUBADDR    .set  0005H     ; 发送控制寄存器 2 （子地址）
SRGR1_SUBADDR   .set  0006H     ; 采样率产生寄存器 1 （子地址）
SRGR2_SUBADDR   .set  0007H     ; 采样率产生寄存器 2 （子地址）
PCR_SUBADDR     .set  000EH     ; 引脚控制寄存器（子地址）
;SPCR1 寄存器组织
*   ---------------------------------------------------------------------------------------------------
*   | 15 |14  13| 12   11| 10    8 |  7  |  6  | 5    4|    3    |    2  |  1  |  0  |
*   ---------------------------------------------------------------------------------------------------
*   |DLB |RJUST|CLKSTP|RESEVED|DXENA|ABIS|RINTM|RSYNCERR|RFULL|RRDY|RRST|
*   ---------------------------------------------------------------------------------------------------
K_DLB           .SET    0b << 15
K_RJUST         .SET    01b << 13
K_CLKSTP        .SET    00b << 11
K_S1_RES        .SET    000b << 8
K_DXENA         .SET    0b << 7
K_ABIS          .SET    0b << 6
K_RINTM         .SET    00b << 4
K_RSYNCERR      .SET    0b  << 3
K_RFULL         .SET    0b  << 2
K_RRDY          .SET    0b  << 1
```

```
K_RRST          .SET        1b
K_SPCR1         .SET        K_DLB|K_RJUST|K_CLKSTP|K_S1_RES|K_DXENA|K_ABIS
|K_RINTM|K_RSYNCERR|K_RFULL|K_RRDY|K_RRST
; SPCR2 寄存器组织
*    ------------------------------------------------------------------------------------------------
*    |15       10 |   9    |   8   |   7   |   6   | 5      4 |      3        |    2      |   1   |   0   |
*    ------------------------------------------------------------------------------------------------
*    | K_S2_RES |FREE |SOFT|FRST|GRST | XINTM|XSYNCERR|XEMPTY|XRDY|XRST|
*    ------------------------------------------------------------------------------------------------
K_S2_RES        .SET        000000b << 10
K_FREE          .SET        0b << 9
K_SOFT          .SET        0b << 8
K_FRST          .SET        0b << 7
K_GRST          .SET        0b << 6
K_XINTM         .SET        00b << 4
K_XSYNCERR  .SET        0b << 3
K_XEMPTY    .SET        1b << 2
K_XRDY          .SET        1b << 1
K_XRST          .SET        1b
K_SPCR2         .SET        K_S2_RES|K_FREE|K_SOFT|K_FRST|K_GRST|K_XINTM
|K_XSYNCERR|K_XEMPTY|K_XRDY|K_XRST
; RCR1 寄存器组织
*    ------------------------------------------------
*    |   15 | 14       8 | 7         5 | 4        0 |
*    ------------------------------------------------
*    | RES1|RFRLEN1|RWDLEN1| RES2   |
*    ------------------------------------------------
K_R1_RES1       .SET        0b << 15
K_RFRLEN1   .SET        0000000b << 8   ; 接收帧长 1 为 1 个字
K_RWDLEN1   .SET        00b << 5        ; 接收字长 1 为 8 比特
K_R1_RES2       .SET        00000b
K_RCR1          .SET        K_R1_RES1|K_RFRLEN1|K_RWDLEN1|K_R1_RES2
; RCR2 寄存器组织
*    ------------------------------------------------------------------------------
*    |    15    | 14      8 | 7         5 |   4         3 |   2   | 1          0 |
*    ------------------------------------------------------------------------------
*    | RPHASE|RFRLEN2|RWDLEN2|RCOMPAND|RFIG| RDATDLY |
*    ------------------------------------------------------------------------------
K_RPHASE        .SET        0b << 15        ; 接收为单相
K_RFRLEN2   .SET        0000000b << 8   ; 接收帧长 2 为 1 个字
K_RWDLEN2   .SET        00b << 5        ; 接收字长 2 为 8 比特
K_RCOMPAND .SET        00b<<3          ; 信号无压扩，先接收 MSB 位
K_RFIG          .SET        0b   <<2        ; 第一次重启发送后接收帧同步脉冲
K_RDATDLY   .SET        01b             ; 1 比特延迟
K_RCR2          .SET
      K_RPHASE|K_RFRLEN2|K_RWDLEN2|K_RCOMPAND|K_RFIG|K_RDATDLY
; XCR1 寄存器组织
```

```
*    ------------------------------------------------
*    | 15   | 14       8 | 7         5 |   4      0 |
*    ------------------------------------------------
*    | RES1|XFRLEN1|XWDLEN1| RES2   |
*    ------------------------------------------------
K_X1_RES1     .SET      0b << 15
K_XFRLEN1     .SET      0000000b << 8   ; 发送帧长 1 为 1 个字
K_XWDLEN1     .SET      00b << 5        ; 发送字长 1 为 8 比特
K_X1_RES2     .SET      00000b
K_XCR1        .SET      K_X1_RES1|K_XFRLEN1|K_XWDLEN1|K_X1_RES2
; XCR2 寄存器组织
*    ------------------------------------------------------------------------------
*    | 15        | 14       8 | 7         5 |   4          3 |   2   | 1          0 |
*    ------------------------------------------------------------------------------
*    | XPHASE |XFRLEN2|XWDLEN2|XCOMPAND|XFIG| XDATDLY |
*    ------------------------------------------------------------------------------
K_XPHASE      .SET      0b << 15        ; 发送为单相
K_XFRLEN2     .SET      0000000b << 8   ; 发送帧长 2 为 1 个字
K_XWDLEN2     .SET      00b << 5        ; 发送字长 2 为 8 比特
K_XCOMPAND .SET      00b<<3          ; 信号无压扩，先发送 MSB 位
K_XFIG        .SET      0b <<2          ; 第一次重启发送后发送帧同步脉冲
K_XDATDLY     .SET      01b             ; 1 比特延迟
K_XCR2        .SET
K_XPHASE|K_XFRLEN2|K_XWDLEN2|K_XCOMPAND|K_XFIG|K_XDATDLY
;PCR 寄存器
*    ------------------------------------------------------------------------
*    | 15    14 |  13   |  12   |  11  |  10   |   9    |  8    |
*    ------------------------------------------------------------------------
*    | reserved | XIOEN | RIOEN | FSXM | FSRM | CLKXM | CLKRM |
*    ------------------------------------------------------------------------
*    ------------------------------------------------------------------------------
*    |    7   |     6    |    5   |   4    |   3  | 2  |   1  |   0   |
*    ------------------------------------------------------------------------------
*     | reserved |CLKS_STAT|DX_STAT|DR_STAT|FSXP|FSRP |CLKXP|CLKRP|
*    ------------------------------------------------------------------------------
K_PCR_RES     .SET   0b << 14
K_XIOEN       .SET   0b << 13
K_RIOEN       .SET   0b << 12       ; DX、FSX 和 CLKX 设置为串口引脚
K_FSXM        .SET   1b << 11       ; 帧同步由 SRGR2 的 FSGMF 位确定
K_FSRM        .SET   1b << 10       ; 帧同步由内部采样率发生器产生，FSR 为输出引脚
K_CLKXM       .SET   1b << 9        ; CLKX 为输出引脚，由内部采样率发生器驱动
K_CLKRM       .SET   1b << 8        ; CLKR 为输出引脚，由内部采样率发生器驱动
K_PCR_RES2    .SET   0000b << 4
K_FSXP        .SET   0b << 3        ; 帧同步脉冲 FSX 高电平有效
K_FSRP        .SET   0b << 2        ; 帧同步脉冲 FSR 高电平有效
K_CLKXP       .SET   0b <<1         ; 发送数据在 CLKX 上升沿被抽样
K_CLKRP       .SET   0b             ; 接收数据在 CLKR 下降沿被抽样
```

```
K_PCR           .SET    K_PCR_RES|K_XIOEN|K_RIOEN|K_FSXM|K_FSRM
  |K_CLKXM|K_CLKRM|K_PCR_RES2|K_FSXP|K_FSRP|K_CLKXP|K_CLKRP
;SRGR1 寄存器组织
*    ------------------------------------------------------------------
*    | 15                    8  | 7                          0 |
*    ------------------------------------------------------------------
*    |          FWID             |          CLKGDV          |
*    ------------------------------------------------------------------
K_FWID          .SET   00000000b << 8      ; 帧同步脉冲宽度（FWID+1）× CLKG
K_CLKGDV        .SET   00101100b           ; 采样率发生器时钟的分频因子
; CLKG＝采样率发生器输入时钟的频率/（CLKGDV+1）
K_SRGR1         .SET   K_FWID|K_CLKGDV
;SRGR2 寄存器
*    ------------------------------------------------------------------------
*    |   15   |  14   |    13    |  12  |  11                0 |
*    ------------------------------------------------------------------------
*    | GSYNC | CLKSP | CLKSM | FSGM |         FPER           |
*    ------------------------------------------------------------------------
K_GSYNC         .SET   0b << 15            ; 采样率发生器时钟同步
K_CLKSP         .SET    0b << 14           ; CLKS 的上升沿产生 CLKG 和 FSG
K_CLKSM         .SET   1b << 13            ; McBSP 采样率发生器的输入时钟源为 CPU 时钟
K_FSGM          .SET   1b << 12            ; FSG 产生帧同步信号
K_FPER          .SET   000011111111b       ; 2.048 MHz Clock/（255+1）=8 kHz
K_SRGR2         .SET   K_GSYNC|K_CLKSP|K_CLKSM|K_FSGM|K_FPER
_serial_init:
STM     SPCR1_SUBADDR,SPSA2
STM     #K_SPCR1_RS,SPSD2          ; XRST=RRST=0，串口复位
STM     SPCR2_SUBADDR,SPSA2
STM     #K_SPCR2_RS,SPSD2          ; FRST=0
STM     RCR1_SUBADDR,SPSA2
STM     #K_RCR1,SPSD2
STM     RCR2_SUBADDR,SPSA2         ; 设置接收字长为 8 比特
STM     #K_RCR2,SPSD2              ; 单相位，无压扩，1 比特延迟
STM     SRGR1_SUBADDR,SPSA2
STM     K_SRGR1,SPSD2              ; 对 SRGR1 的配置见前面对 K_SRGR1 的说明
STM     SRGR2_SUBADDR,SPSA2
STM     K_SRGR2,SPSD2              ; 对 SRGR2 的配置见前面对 K_SRGR2 的说明
STM     PCR_SUBADDR,SPSA2
STM     K_PCR,SPSD2                ; 对 PCR 的配置见前面对 K_PCR 的说明
STM     XCR1_SUBADDR,SPSA2
STM     #K_XCR1,SPSD2
STM     XCR2_SUBADDR,SPSA2         ; 设置发送字长为 8 比特
STM     #K_XCR2,SPSD2              ; 单相位，无压扩，1 比特延迟
STM     SPCR2_SUBADDR,SPSA2
STM     #K_SPCR2_RS_ONE,SPSD2      ; 退出复位状态
STM     SPCR2_SUBADDR,SPSA2
STM     #K_SPCR2_RS_TWO,SPSD2
```

```
STM     SPCR1_SUBADDR,SPSA2
STM     #K_SPCR1,SPSD2
STM     SPCR2_SUBADDR,SPSA2
STM     #K_SPCR2,SPSD2
RET
```

10.3 DSP 系统的 BOOT 设计

DSP 脱机系统设计的核心是了解 DSP 芯片提供的 BOOT 方式及其实现方法。所谓 BOOT，就是“自引导”或“自启动”的意思，就是加电复位后首先运行固化在 DSP 芯片内部的引导程序，通过该程序将存放在外部的用户程序装载到片内或片外快速 RAM 中。BOOT 结束后，将程序跳转至用户程序入口处，开始运行用户程序。

下面分别介绍 TMS320C5x、TMS320C54x、TMS320C55x、TMS320C6x 等芯片的 BOOT 设计。

10.3.1 TMS320C50 的 BOOT 设计

在 TMS320C5x 系列 DSP 芯片中，TMS320C51 内部具有 8K 字的掩膜 ROM，使用片内的掩膜 ROM 需要将应用程序交由 TI 帮助完成掩膜工作，因此在研究开发阶段一般不用内部的掩膜 ROM。TMS320C50 虽然没有掩膜 ROM，但其内部具有 2K 字的 BOOT ROM，工作在微计算机方式下可以启动 BOOT ROM 中的引导程序，将用户程序从 EPROM、串行口或 I/O 口引导到 10K 字的内部 RAM 或外部的高速 RAM 中全速运行。下面介绍一下 C50 的程序引导方法。

TMS320C50 硬件复位后采样 MP/$\overline{\text{MC}}$引脚，如果为低，则运行片内的引导程序。引导程序首先进行初始化，包括关闭中断（INTM=0），将内部的双存取 RAM 块 B0 映射至程序空间（CNF=1），使内部的单存取 RAM 块在程序空间中有效（RAM=1,OVLY=0），程序和数据空间都选择 7 个等待状态，数据空间 8000H－0FFFFH 的 32K 字设置为全局存储器。用户程序装载完后，撤销全局存储器设置。

C50 程序引导共有并行 EPROM、并行 I/O、串行口和 Warm 四种方式，其中前三种又有 8 位和 16 位两种。引导方式是通过读取全局数据空间的 0FFFFH 单元的低 8 位来确定的，其内容和引导方式的对应关系如表 10-1 所示。

表 10-1　C50 程序引导方式

方　式	0FFFFH 单元低 8 位内容
8 位串行口	X x x x 0 0 0 0
16 位串行口	x x x x 0 1 0 0
8 位并行 I/O	x x x x 1 0 0 0
16 位并行 I/O	x x x x 1 1 0 0
8 位并行 EPROM	SRC　0 1
16 位并行 EPROM	SRC　1 0
Warm	ADDR　1 1

表 10-1 中，x 表示无效，SRC 表示并行方式的 6 位页地址，ADDR 表示 Warm 方式的 6 位页地址。

在四种引导方式中，并行 EPROM 是最常用的引导方式，该引导方式将存储在 EPROM 中的程序引导到内部或外部的程序存储器。其中，EPROM 映射在全局数据空间中，可以是 16 位宽度，也可以是 8 位宽度。引导程序在装载用户程序时需要确定以下几个参数：（1）存放用户程序的 EPROM 起始地址；（2）用户程序的目的地址；（3）用户程序的长度。EPROM 本身的起始地址是由 6 位 SRC 左移 10 位而得，即

```
15                    10 9 8 7 6 5 4 3 2 1 0
<—    SRC        —> 0 0 0 0 0 0 0 0 0 0
```

用户程序的目的地址和长度对几种引导方式而言是类似的，其中目的地址由第一个 16 位字确定，而长度则由第二个 16 位字确定。

对于 16 位 EPROM 方式，EPROM 的第一个地址存放目的地址，第二个地址存放程序长度，如果程序长度为 N，则其内容为 N－1。接下去依次存放用户程序的第一个字，第二个字，……，第 N 个字。对于 8 位 EPROM 方式，EPROM 的 8 位数据线与 DSP 芯片的低 8 位数据线相连，两个连续的字节组成一个 16 位字，其中高字节在前，低字节在后。与 16 位方式一样，前两个字分别为目的地址和程序长度，程序长度按 16 位字计算。接下去便是用户程序，用户程序也是按照高字节在前，低字节在后的顺序存放。程序引导完后，跳至目的地址开始执行用户程序。由于用户程序是从全局数据空间装载至程序空间，因此 EPROM 的片选应是总线请求线$\overline{\text{BR}}$和数据选择线$\overline{\text{DS}}$的与信号，图 10-1 是一种可能的采用 8 位 EPROM 实现程序引导的接口方法。

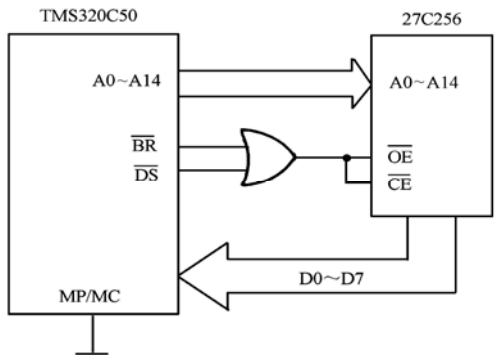


图 10-1　利用 EPROM 引导用户程序

在串行口引导的两种方式中，串行口控制寄存器 SPC 分别设置为 0xxF8H（8 位）和 0xxFCH（16 位），RRST 和 XRST 位设置为 1，FSM＝1 为帧同步方式，FSR 由外部提供同步时钟。FO 位根据 8 位和 16 位作相应的设置。用 XF 线表示 C50 是否准备好接收数据，复位后 XF 为高，XF 变低表明 C50 已准备好接收数据。FSR 引脚在 XF 变低前不应有脉冲。与 EPROM 方式一样，从串行口接收到的第一个字和第二个字分别为用户程序的目的地址和长度，随后便是用户的程序代码。采用 8 位串行方式时，一个字由两个字节组成，高字节在前，低字节在后。用户程序引导完后，跳至目的地址开始执行用户程序。

C50 也可以从并行的 I/O 口引导程序。I/O 口的地址固定为 50H，可以是 16 位，也可以是 8 位。采用 XF 和$\overline{\text{BIO}}$线作为 C50 与主机之间的握手线。主机发送数据和 C50 接收数据的过程为：主机将$\overline{\text{BIO}}$置高请求数据发送，C50 将 XF 置低表示已准备好接收数据，主机将数据送至 I/O 口并将$\overline{\text{BIO}}$置低表示数据有效，C50 从 50H 口读入数据并将 XF 置高表示数据接收完毕，一次数据传递过程结束。其他与前两种方式相同。

Warm 方式严格来说并不是程序引导方式。当程序已经用其他方式（如 DMA）传送至程序存储器而只需运行用户程序时，就采用这种方式。Warm 方式实际上就是确定用户程序的起始地址，并跳至该地址执行。用户程序的起始地址由 ADDR 左移 10 位得到。

10.3.2 TMS320C54x 的 BOOT 设计

TMS320C54x DSP 芯片一般都在片内设置有 BOOT 程序（Bootloader），其主要作用是在开机时将用户程序从外部装入到程序存储器。

当 TMS320C54x DSP 硬复位后，如果 MP/$\overline{\text{MC}}$脚为低电平，则从 FF80h 地址开始运行。在该地址处放置了一个跳转指令，跳转到厂商固化在 ROM 中的 Bootloader 程序的开始地址。Bootloader 程序在正式下载执行代码前，会首先设置 CPU 的寄存器：禁止所有的可屏蔽中断（INTM=1）；将内部 RAM 映射到程序/数据区（OVLY=1）；对所有的数据区和程序区的读写插入 7 个等待状态；将外部存储区的每个块（Block）大小设置为 4K 字。

然后，Bootloader 程序根据中断、$\overline{\text{BIO}}$、XF 等多种信号来决定 DSP 将要采用的 BOOT 模式，并利用该模式将外部存储器中的代码搬移到指定的片内存储区。在以上过程完成后，Bootloader 程序将 XPC 和 PC 值设置为搬移入的执行代码的首地址，开始运行用户程序。

TMS320C54x DSP 芯片提供了多种 BOOT 方式，通常包括：并行口方式、串行口方式、HPI 方式、I/O 方式等。不同 DSP 芯片的 BOOT 方式略有不同。本节对 TMS320C54x DSP 芯片的不同 BOOT 方式进行简要介绍。

1．并口BOOT方式

并口 BOOT 方式包括 8-bit 和 16-bit 两种。对于不同的 TMS320C54x，并口 BOOT 的被选择顺序不同，如 C541 和 C542 芯片是在 I/O BOOT 方式判断之后，C5409 是在串口 EEPROM BOOT 方式判断之后，C5410 是在 HPI BOOT 方式判断之后，等等。

并口 BOOT 方式的判断都是根据 I/O 空间的 FFFFh 地址读入的数据来决定的。C541 和 C542 芯片通过判断读入的二进制数据最后两个比特位来决定所采用的 BOOT 方式：当最后两个比特位为 01b，则采用 8-bit 并口 BOOT 方式；若为 10b，则采用 16-bit 并口 BOOT 方式。从 C548 和 C549 芯片开始，并口 BOOT 的检测方式改为如图 10-2 所示的流程。

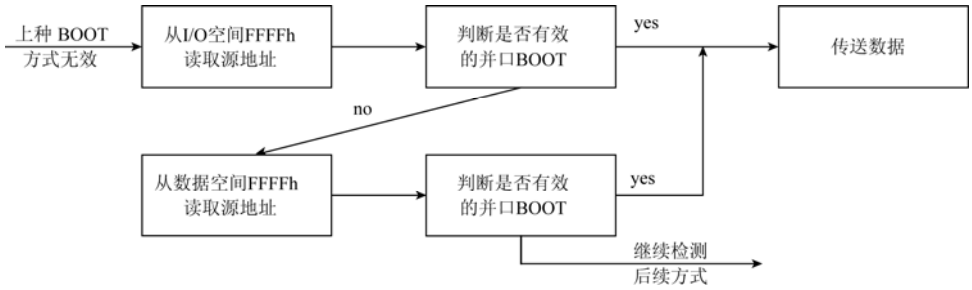


图 10-2　C548/C549 的并口 BOOT 检测方式流程图

图中有效 BOOT 方式的检测实现如下：利用所取得的源地址进行寻址读取数据，若得到的数据为 08AAh，则表明并口 BOOT 有效，将以 8-bit 并口方式进行后续 BOOT 工作；若为 10AAh，则表明并口 BOOT 有效，将以 16-bit 并口方式进行后续 BOOT 工作。否则，判断为并口 BOOT 方式无效，继续下一种 BOOT 方式的检验。

在这种并口 BOOT 检测方式下，如果系统没有采用并口 BOOT 的方式，则建议采用一定的手段，如将外部数据总线 D0 上拉等方式，来避免并口 BOOT 方式的误检测。

由于 BOOT 已经设定好相应的数据存放格式，因此，在 EPROM 中组织好数据就成为关

键。下面我们以 C548 芯片的 8-bit 并口 BOOT 方式为例，做以下假设来具体说明在这种方式中，存放在外部 EPROM 中的数据组织方法：存放在 EPROM 中的控制代码和用户代码的首地址为外部数据区的 8000H、等待状态数为 7 个、bank 长度为 64 K 、程序执行的入口点地址为 2000H、程序代码的长度为 400H、用户代码存放在片内程序区的首地址为 2000H，则 EPROM 中的数据组织如表 10-2 所示，称为 BOOT 表。

表 10-2　EPROM 的数据组织

数据区地址（Hex）	内　　容（Hex）	含　　义
8000	08	8 位 BOOT 标识（H）
8001	AA	8 位 BOOT 标识（L）
8002	7F	SWWSR（H）
8003	FF	SWWSR（L）
8004	F8	BSCR（H）
8005	00	BSCR（L）
8006	00	程序入口 XPC（H）
8007	00	程序入口 XPC（L）
8008	20	程序入口地址（H）
8009	00	程序入口地址（L）
800A	04	程序块长度（H）
800B	00	程序块长度（L）
800C	00	存放目标 XPC（H）
800D	00	存放目标 XPC（L）
800E	20	存放目标地址（H）
800F	00	存放目标地址（L）
8010	XX	程序代码 1（H）
8011	XX	程序代码 1（L）
…	…	…
880E	XX	程序代码 N（H）
880F	XX	程序代码 N（L）
8810	00	块结束标志（H）
8811	00	块结束标志（L）
…	…	…
FFFE	80	代码存放首地址（H）
FFFF	00	代码存放首地址（L）

表 10-2 中括号中的 H 表示高 8 位，L 表示低 8 位。字节需放置在数据线的低 8 位。

若采用 16-bit 的并口 BOOT 方式，则外部 EPROM 中的数据以 16-bit 的方式存放。此时，BOOT 表中的数据存放格式为 16-bit 形式，而存放的 BOOT 表数据顺序不变。

需要注意的是，不同 TMS320C54x DSP 芯片的 BOOT 表的组织形式略有不同，更详细的内容可参考不同芯片的具体参考手册。

我们在使用 TMS320LC548/9 DSP 芯片的 8 位 BOOT 时，发现了内部 BOOT 程序存在着以下缺陷：

（1）TMS320LC548 在读取外部两个 8 位字节组装成一个 16 位字时，没有考虑到高 8 位数据的随机性，而要求高 8 位数据为 0，这给用户带来不便。如果外部高 8 位数据不用的话，则必须拉低；如果系统中需要高 8 位数据线接至其他外设，那要求用户怎么保证在 BOOT 阶

段使 D8~D15 数据线为低电平？

（2）TMS320LC549 的内部 8 位 BOOT 程序在 TMS320LC548 的基础上作了改进，基本上不要求高 8 位数据为 0。但遗憾的是，TMS320LC549-66 仍然存在以下缺陷：

在判断块是否结束时，BOOT 程序如下：

```
LD      *AR1+,8,A        ；AR1 指向的地址内容左移 8 位→A
MVDK    *AR1+,AR3        ；AR1 指向的地址内容→AR3
ANDM    #0FFH,@AR3       ；AR3 内容高 8 位清零
OR      @AR3,A           ；组装成一个字
BCD     ENDBOOT,AEQ      ；如果 ACC=0，BOOT 结束
```

我们来看一下这段程序的执行结果：

程序执行前，AR1=8810H，ACC=0，设读入的外部数据的高 8 位为 FFH：

第 1 句执行后，ACC=00FF0000H;

第 2 句执行后，AR3=FF00H;

第 3 句执行后，AR3=0000H

第 4 句执行后，ACC=00FF0000H;

第 5 句执行后，判断 ACC 不为 0，因此 BOOT 并未结束。

所以，这同样要求高 8 位数据线在 BOOT 阶段为低电平。

由于外部并行 BOOT 是在外部数据区进行的，而数据区的前 32K 地址已经被内部 RAM 所占用，因此实际能够寻址的范围仅为 8000H~FFFFH，共 32K。如果需要使用超过 32K 的外部 BOOT，可以采用以下 BOOT 法。

第 1 步：内部 BOOT。利用片内 BOOT 程序将自己编制的 BOOT 程序从外部数据区搬移至内部 RAM 中。

第 2 步：用户 BOOT。内部 BOOT 完成后，开始执行自己的 BOOT 程序。利用 DSP 的扩展寻址方法，自己编制的 BOOT 程序既可从数据区读取代码，也可以在外部扩展的程序区读取代码。

第 3 步：用户 BOOT 完成后，跳至用户程序开始运行。

2．HPI BOOT方式

TMS320C54x DSP 芯片复位后，在 BOOT 方式选择阶段首先检查 INT2 的状态。Bootloader 程序首先将 HINT 清零，随后检测 IFR 中的 INT2 标识是否为 1（有效）。如果为 1，则开始 HPI 的 BOOT 模式，否则，HPI 的 BOOT 模式被跳过。以下两种方式可以在复位后有效设置 INT2 的标识：

（1）将 HINT 脚连接到 INT2 输入端；

（2）在 DSP 获得 Reset 中断矢量后的 30 个 CPU 时钟周期内在 INT2 输入端提供有效的中断信号。

一旦进入 HPI BOOT 方式，DSP 等待外部主机将执行代码写入到特定的存储位置，并在外部主机将 DSP 唤醒后，开始执行写入的执行代码。

对于 C541、C542 等早期的 TMS320C54x DSP 芯片来说，由于在 HPI BOOT 过程中，写入的执行代码大小和执行首地址都无法被 Bootloader 程序得知，因此，在 BOOT 结束后，将自动从 1000h 地址开始执行。对于 C5410 芯片，这个执行首地址为 000 2000h。

对于 C5409 芯片，在 HPI 模式下，DSP 会在复位后立即将 0x7Fh 清零，并不断检测此处

的值。在 HPI BOOT 进程最后，主机需将 DSP 的执行首地址写入 0x7Fh 地址。Bootloader 程序一旦发现这个变化，立即跳转到该地址开始执行。

对于 C5402A、C5409A、C5410A 和 C5416 芯片，在复位后，Bootloader 程序将数据空间地址 007Eh 和 007Fh 都置为“0”，并用它们作为主机结束 HPI 下载的软标识；Bootloader 程序在外部主机加载期间将一直检测片内数据区 RAM 地址 007Fh。主机加载完毕后，必须将执行入口地址的 XPC 和 PC 值分别写入 DSP 片内的数据区地址 007Eh 和 007Fh，且写入的 PC 值不能为 0。随后 Bootloader 程序将跳转到执行入口地址，开始执行代码。

3．串口BOOT方式

串口 BOOT 方式包括标准串口 BOOT、8-bit EEPROM 串口 BOOT 和 TDM 串口 BOOT 三种。

（1）标准串口 BOOT

在标准串口 BOOT 模式下，Bootloader 程序将串口初始化为 TMS320C54x 标准串口配置，并将 XF 置低，向外部设备表明串口已准备好接收数据。DSP 随后检测 IFR，以确定 McBSP0 和 McBSP2 口哪个有数据输入。一旦串口被确定，随后读该串口以取得 BOOT 表的内容。

XF 的低电平信号发出后，串口主机设备可以开始通过 McBSP0 或 McBSP2 口发送 BOOT 表的信息。这两个 McBSP 口的一个接收到信息后，Bootloader 程序会利用产生的中断标识来分辨哪个 McBSP 口被用来进行 BOOT。由于 McBSP0 口被指定执行 16-bit 的 BOOT 模式，McBSP2 口被指定执行 8-bit 的 BOOT 模式，因此此时的 BOOT 模式也被确定。

如果此时传送来的字是该 McBSP 口对应 BOOT 方式的 BOOT 表首关键字（08AAh 或 10AAh），则就使用该串口的特定模式进行 BOOT，而不会再进行其他 BOOT 模式的检测。如果两个 McBSP 口都没有接收到数据，Bootloader 程序将继续进行后续 BOOT 模式的检测。

在 C548/C549 中，Bootloader 程序在读取关键字后将读取串口配置字，以重新配置串口。在 C5410 上，串口在读取 BOOT 表的过程中不会被重新配置。为保持十六进制文件的 BOOT 表格式的一致性，C5410 将忽略串口 BOOT 表中的串口配置块部分。

为保证操作正确，对于 C548/C549 芯片来说，串口需满足以下两个条件：

- 串口时钟速率不大于 1/8 的 CLOCKOUT 时钟速率；
- 在上一个字的最后和下一个帧同步信号之间最少需要 8 个串口时钟的间隔。

对于 C5410、C5416 等芯片来说，串口 BOOT 需满足以下两个条件：

- 参考接收时钟 BCLKR 由外部提供，且速率不能超过 C5410 的 CPU 时钟的 1/2；
- 每个字之间最少需要有 40 个 CPU 时钟周期的延迟。可通过降低串口时钟频率，或在传送的字间增加额外时钟脉冲的方法来达到目的。

（2）8-bit EEPROM 串口 BOOT

这种模式被 C5409、C5402A、C5409A、C5410A、C5416 等芯片采用。

该模式的检测要使用到 INT3 标识。若在 DSP 获取 RESET 中断矢量后的 30 个 CPU 时钟周期内，检测到 INT3 脚的电平由高到低产生变化，则 Bootloader 程序认为 BOOT 表将通过连接在 McBSP2 口的 8-bit 串口 EEPROM 送入。DSP 将首先读取 EEPROM 中地址 0 的数据，若此时读取的为有效的关键字（08AAh），则继续读取剩下的 BOOT 表内容。若 INT3 无效，或没有发现关键字，则继续检测下一 BOOT 模式。

INT3 脚电平的变化可以由外部逻辑电路产生，也可以利用 McBSP2 口中的 BDX2 脚来完成该功能。BDX2 脚在复位后的前几个周期内会自动产生从高到低的触发信号，因此可以将该

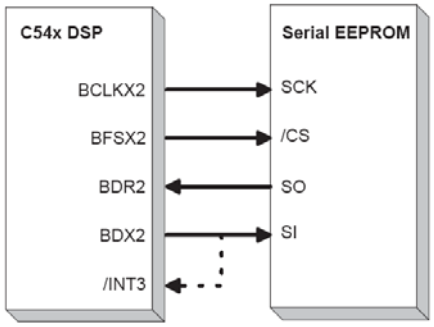


图 10-3　C54x DSP 与串口 EEPROM 间的连接

脚与 INT3 脚相连，以完成该模式的选择功能。

在这种 BOOT 模式中，DSP 将 McBSP2 的时钟速率设置为 CLKOUT/250，即对于 100MHz 主频则为 400KHz。这种低速可以适用于大部分的串口 EEPROM。McBSP2 配置为时钟一停止（clock-stop）工作模式，时钟和帧信号都由 DSP 内部产生，CLKSTP＝3，CLKXP=0，CLKXM=1，作为 SPI 的主口。该模式的相关时序在 DSP 的手册中可见。此时 EEPROM 必须为 4 线串口的从口，采用 16-bit 地址。DSP 与 EEPROM 间的连接如图 10-3 所示。

对于每次读，McBSP2 传送一个 32-bit 的包，其中包括 8-bit 的读指令（03h）和 16-bit 的地址，最后是一个总线锁定字节。EEPROM 会忽略最后的字节，并于这个时刻在 SO 上将相应地址的数据发送出去，如图 10-4 所示。

串口 EEPROM BOOT 模式可用于装载多个块的内容到可用的程序空间，包括扩展的程序空间。由于只有 16-bit 的地址，因此最大只能装载 64K 字节，或 32K 的 16-bit 字，限制了装载的代码量。

（3）TDM 串口 BOOT

这种模式在 C541 和 C542 中，作为串口 BOOT 方式之一定义。

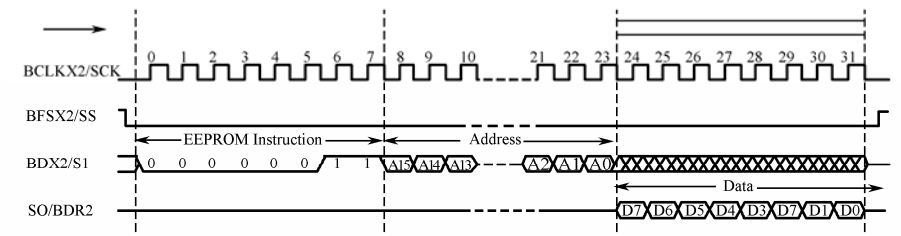


图 10-4　C54x DSP 与串口 EEPROM 间的数据传送时序

除此外，在 C548/C549 中有特殊的定义。在这种 BOOT 方式中，是采用 BSP1 口进行数据的交流，并限定为 16-bit 模式。该 BOOT 模式的检测需利用 INT0、INT1、INT3 三个中断是否有效来决定，并根据三个中断的有效组合，来决定所采用的 TDM 通道，如表 10-3 所示。

Bootloader 利用中断值来配置 TDM 的 TRTA 寄存器，将 TFSX 和 TCLKX 设置为输入。随后将 XF 脚置为低以指示 TDM 口已准备好接收数据。所接收到的第一个字必须为 10AAh，否则说明该模式无效。

表 10-3　TDM 口 BOOT 方式的通道选择

［INT3， INT1，INT0］	［0，0，1］	［0，1，0］	［0，1，1］	［1，0，0］	［1，0，1］	［1，1，0］	［1，1，1］
Channel	2	3	4	5	6	7	8

4．I/O口BOOT方式

I/O 口的 BOOT 方式为 DSP 提供了一种异步的并口 BOOT 方式。

对于 C541/C542 芯片，仍通过从 I/O 空间的 FFFFh 地址所读取的数来判断是否为 I/O 口的

BOOT 模式。而对于其他 TMS320C54x DSP 芯片，I/O 口的 BOOT 模式是通过外部主机将 DSP 芯片的 $\overline{\text{BIO}}$ 脚设为低来启动的。

在 I/O 口的 BOOT 过程中，DSP 芯片与外部主机间通过 XF 和 $\overline{\text{BIO}}$ 作为握手信号来通信。$\overline{\text{BIO}}$ 变低后，DSP 从 I/O 空间的地址 0h 读数据，再将 XF 置高以指示读取完成，随后将输入数据写入目标地址。DSP 随后将 XF 置低，等待 $\overline{\text{BIO}}$ 变高再变低后读取下一个字，如图 10-5 所示。这种握手方式可以让慢速的主机与高速的 DSP 间进行通信。

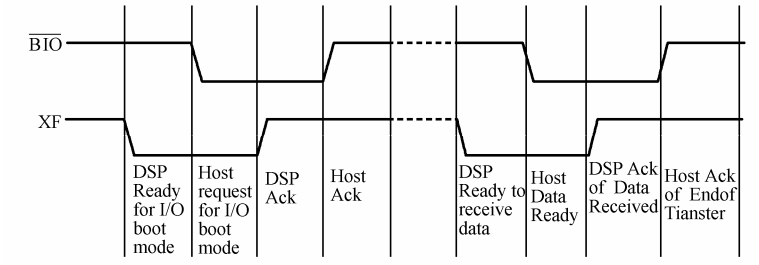


图 10-5　I/O 口 BOOT 的握手时序

在 XF 上升沿和写操作之间至少要有 10 个 CPU 时钟周期的延迟，以便在目标地址是外部存储区时，主机有时间将数据缓冲区关闭。注意，DSP 芯片只能在 XF 为高的期间驱动外部总线。

I/O 口的 BOOT 可以 8-bit 方式，也可以 16-bit 方式进行。当采用 8-bit 方式时，低 8 位数据线有效，16-bit 字的高位字先发送，低位字后发送。

对于 C541/C542，传送的前两个字必须为目的地址和代码长度，而对于其他 TMS320C54x DSP 芯片，最先接收的则必须是关键字（08AAh 或 10AAh）。

5．BOOT表的建立

只要采用非 HPI 口的 BOOT 方式，都需要用到 BOOT 表格式进行数据组织。

对于 C541/C542 芯片，其 16-bit 模式的 BOOT 表的格式如表 10-4 所示，8-bit 模式的 BOOT 表的格式如表 10-5 所示。其中，表 10-4 的数据是以 16-bit 的字形式存放，表 10-5 中的数据是以 8-bit 的字节形式存放。

对于其他的 C54x DSP 芯片，其 BOOT 表的数据存放组织如图 10-6 所示。

为得到 BOOT 表，必须使用代码格式转换器。代码格式转换器的作用是将链接器生成的.out 文件（COFF 文件格式）转换为其他文件格式，如 Intel 的 Hex 文件格式、Tektronix 的文件格式或 TI-tagged 文件格式，以便于烧制 EPROM，因为大多数的 EPROM 编程器都不接受 COFF 文件格式。

表 10-4　C541/C542 的 16-bit BOOT 表格式

目的地址
代码长度 N−1
代码字 1
……
代码字 N

表 10-5　C541/C542 的 8-bit BOOT 表格式

目的地址高 byte
目的地址低 byte
代码长度 N−1 的高 byte
代码长度 N−1 的低 byte
代码字 1 的高 byte
代码字 1 的低 byte
……
代码字 N 的高 byte
代码字 N 的低 byte

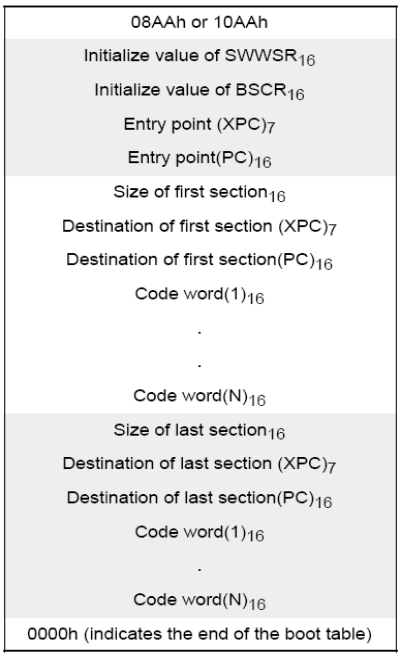


图 10-6　带关键字的 BOOT 表

在 TMS320C5000 代码生成程序包中提供了 Hex500.exe 应用程序，下面简单介绍该程序的使用方法。

代码格式转换的命令格式为：

Hex500　[-选项][COFF 输入文件[输出文件 1[输出文件 2]]]

其中，选项有四种：

-x　输出为 Tektronics hex 格式
-i　输出为 Intel hex 格式
-w　输出为 Intel word 格式
-t　输出为 Ti-tagged 格式

如果不指定选项，则默认的输出文件格式为 Tektronics hex 格式。输入文件若不指定扩展名，则默认的扩展名为 .obj。若不指定输出文件名，则输出文件的扩展名用.hi, .lo, .hex 和.tag 等。例如：

（1）Hex500　- i　test.out

生成 Intel hex 格式的两个文件 test.hi 和 test.lo，其中 test.hi 包含高字节部分，test.lo 包含低字节部分。

（2）Hex500　-x　test.out

生成 Tektronics 格式的两个文件 test.hi 和 test.lo，分别包含高字节和低字节。

（3）Hex500　-w　test.out

生成 Intel 字格式的一个文件 test.hex。

（4）Hex500　-t　test.out

生成 Ti-tagged 格式的一个文件 test.tag。

对于 C548/C549/C5409/C5410/C5416 等 DSP 芯片，在编译中必须选用-v548 选项，否则所产生的 OUT 文件在格式转换后所得到的 BOOT 表是 C541/C542 的格式，由于缺少关键字，将在 BOOT 过程中产生错误。

10.3.3　TMS320C55x 的 BOOT 设计

TMS320C55x 系列 DSP 芯片加电后，Bootloader 程序首先对 DSP 进行初始化配置，其配置内容如表 10-6 所示。

表 10-6　C5503/C5507/C5509/C5509A 的 Bootloader 程序初始化配置

配置对象	初始化配置
堆栈寄存器	数据堆栈寄存器 SP 指向地址 000090h，系统堆栈寄存器 SSP 指向地址 000080h
堆栈配置	设置为默认的 32-bit 堆栈模式
可寻址寄存器	将地址 000060h 和 000061h 的两个字保留为入口地址的暂存区
符号扩展	ST1_55 中的 SXMD 比特设置为 0，以禁止符号扩展功能；执行代码复制完毕后，代码执行前，将 SXMD 设置为 1
中断	ST1_55 中的 INTM 比特设置为 1，以禁止中断
兼容模式	ST1_55 中的 54CM 比特设置为 1，允许兼容模式

随后，根据片上的 BOOT 模式选择 BOOTM[0:3]的配置，选择对应的 BOOT 方式，具体的

选择如表 10-7 所示。对于 C5503/C5507/C5509/C5509A 芯片，BOOTM[0:3]的引脚与通用 IO 引脚共用，其中，BOOTM3 与 IO0 共用，BOOTM2 与 IO3 共用，BOOTM1 与 IO2 共用，BOOTM0 与 IO1 共用。对于 C5510/C5510A 芯片，BOOTM3 为一个独立的引脚。IO4 则在某些 BOOT 模式中用作握手信号的输出。BOOT 结束后，载入的程序可以改变 IO[0:4]的功能。

BOOT 结束后，Bootloader 程序转向执行代码的首地址，开始程序的执行。

1．从外部的异步存储器直接执行代码

这种模式下，Bootloader 将 EMIF 配置为 16-bit 异步存储器模式，并从外部程序区字节地址 40 0000h 开始执行。因此此处存放的必须是执行代码，而不能是 BOOT 表。为适应慢速的存储器，Bootloader 将 READ SETUP、READ STROBE、READ HOLD 和 READ EXTENDED HOLD 参数设置为最大。

2．并口EMIF的BOOT

这种模式下，Bootloader 从外部的 8-bit 或 16-bit 异步存储器中读取 BOOT 表。

并口 EMIF 模式从位于 CE1 空间的地址 20 0000h 开始读取 BOOT 表。BOOT 表中包含执行代码的入口地址。为适应慢速的存储器，Bootloader 将 READ SETUP、READ STROBE、READ HOLD 和 READ EXTENDED HOLD 参数设置为最大。若这种设置让数据传送太慢，可以通过设置端口寄存器的方式来修改。具体操作可参考用户手册。但要注意的是，在 BOOT 过程中调整时间参数，可能导致 BOOT 失败。

对于 EMIF 控制寄存器的调整也可能导致一些潜在的问题。因此，BOOT 表中入口处的 EMIF 重配置参数必须跟随不小于 10 个周期的延迟。

在这个 BOOT 模式中，IO4 在 BOOT 进程的开始为低。当 BOOT 表中的延迟被执行时，IO4 变为高；延迟结束，IO4 返回为低。BOOT 进程结束后，IO4 将变为高。IO4 的这种变化可以用于辅助控制外部的异步存储器，以便于 BOOT 进程的进行。

3．EHPI的BOOT模式

这种 BOOT 模式无需 BOOT 表。在这种模式下，复位后，DSP 根据 BOOTM 的选择进行相应的配置。配置完成后，IO4 被置为低，以标识本芯片已准备好从主机接受数据。随后，主机可以将执行代码或数据传送入 CPU 存储区，并在数据传送结束后通知 CPU 开始执行代码。

对于 C5503/C5507/C5509/C5509A 芯片，一般情况下，主机可以使用字地址小于 00 4000h 的地址空间，这个地址空间包括片内 DARAM 块 0-3。因为其中一些 DARAM 存储区被 Bootloader 程序使用，因此建议通过 EHPI 口传送的执行代码放置空间限制在字地址 00 0100h~00 3FFFh 之间。对于 C5510/C5510A 芯片，这个空间为 00 0030h~0F FFFFh。

对于 C5503/C5507/C5509/C5509A 芯片，执行代码的入口地址则存放在 DARAM 的字地址 0060h（高位）和 0061h（低位）。其中，低 24-bit 位为执行代码的入口字节地址，高 8-bit 为停止标识位，如图 10-7 所示。在 EHPI 传送代码期间，CPU 一直检测停止标识位，直至其值为非 0 时，表明 BOOT 进程结束，此时 CPU 跳转到执行代码的入口地址开始执行。因此，字地址 0060h 的内容应在 BOOT 流程的最后写入。需注意的是，EHPI 主机地址使用的是字地址，而 DSP 的 CPU 采用的是字节地址。

对于 C5510/C5510A 芯片，入口地址固定于内部 SARAM 块的 01 0000h 字节地址。当主

机完成数据传送后，将 DSP 的 HPIC 寄存器中的 RESET 比特置为 1，DSP 随后从 01 0000h 字节地址开始执行程序。

表 10-7　C5503/C5507/C5509/C5509A 的 BOOT 模式选择

BOOTM[3:0]				Device(s)			Boot Source
IO.0	IO.3	IO.2	IO.1	VC5503	VC5507/ VC5509A	VC5509	
0	0	0	0	–	–	–	Reserved
0	0	0	1	Yes	Yes	Yes	Serial EEPROM (SPI − 24-bit address) boot from McBSP0
0	0	1	0	No	Yes	Yes	USB
0	0	1	1	Yes	Yes	–	I2C EEPROM
0	1	0	0	–	–	–	Reserved
0	1	0	1	Yes	Yes	Yes	EHPI (multiplexed mode) boot
0	1	1	0	Yes	Yes	Yes	EHPI (non-multiplexed mode) boot
0	1	1	1	–	–	–	Reserved
1	0	0	0	Yes	Yes	Yes	Execute from 16-bit external asynchronous memory
1	0	0	1	Yes	Yes	Yes	Serial EEPROM (SPI − 16-bit address) boot from McBSP0
1	0	1	0	Yes	Yes	–	Parallel EMIF boot (8-bit asynchronous memory)
1	0	1	1	Yes	Yes	Yes	Parallel EMIF boot (16-bit asynchronous memory)
1	1	0	0	–	–	–	Reserved
1	1	0	1	–	–	–	Reserved
1	1	1	0	Yes	Yes	Yes	Standard serial (16-bit data) boot from McBSP0
1	1	1	1	Yes	Yes	Yes	Standard serial (8-bit data) boot from McBSP0

4．标准串口BOOT

标准串口 BOOT 模式采用 McBSP0 口来传送执行代码，可以是 8-bit 模式，也可以是 16-bit 模式。McBSP0 口由 Bootloader 程序配置为：

RPHASE＝0b；
RFRLEN1＝0000000b；
RJUST＝00b；
RDATDLY＝01b；
RWDLEN1＝000b（8-bit 模式）或 010b（16-bit 模式）；
CLKR0 和 FSR0 配置为外部输入。

配置完成后，DSP 的 IO4 被置为低，以标识 DSP 芯片已准备好接受数据。随后开始执行代码的接收。在此过程中，IO4 脚作为握手信号使用：串口准备好从外部接收数据，则 IO4 为低，否则为高。

为顺利进行 BOOT，必须满足以下两个条件：

- 外部提供的串口接收时钟速率需低于 DSP 的 CPU 时钟速率的 1/8；
- 传送的字之间必须有适当的延迟。

5．SPI EEPROM的BOOT模式

这种 BOOT 模式同样使用串口 McBSP0，此时该串口被配置成时钟－停止（clock-stop）

工作模式。传送数据模式可以是 16-bit 和 24-bit 中的一种。在这种 BOOT 模式中，DSP 作为主机工作，其 CLKX0 的周期=244×（DSP 输入时钟周期），IO4 用于作为 EEPROM 的片选信号 CS。图 10-8 所示为该 BOOT 模式的最小连接方式。

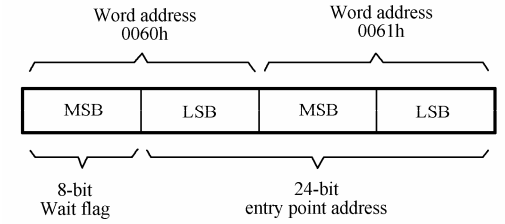


图 10-7　执行代码入口地址

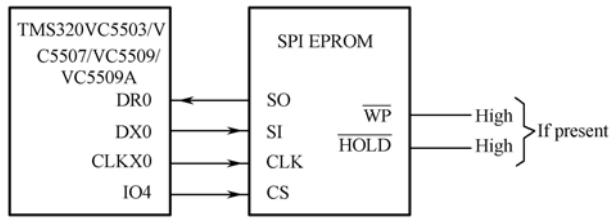


图 10-8　SPI EEPROM 与 DSP 的连接示意图

6．I^2C EEPROM的BOOT模式

C5503/C5507/C5509/C5509A 芯片支持这种模式，C5510/C5510A 芯片不支持。这种模式可以让 DSP 利用外部具有 I^2C 总线的存储器进行 BOOT。但需要满足以下三个条件：

- 存储器件需符合 Philips I^2C 总线规范 V2.1，从地址为 50h；
- 内部地址采用两个字节表示，即范围为 64K 字节；
- 存储器件需具备地址自动增加的功能。

10.3.4　TMS320C6x 的 BOOT 设计

对于 TMS320C6x 系列的 DSP 芯片，在加电复位后，DSP 将首先初始化寄存器，随后进行 BOOT 进程。对于 C620x/C670x，BOOTMODE[4:0]的初始值决定了 BOOT 模式。对于 C621x/C671x，HPI 上的上拉/下拉电阻配置状态与 CLKMODE0 的输入信号一起，用于确定 BOOT 模式。对于 C64x，BOOT 模式通过特定 EMIF 数据总线上的上拉/下拉电阻的配置来决定。例如，C6414 的 BOOT 模式由 HPI 数据线引脚 HD5 和 EMIF 地址总线（BEA[20:1]）上的上拉/下拉电阻决定；C6415 的 BOOT 模式由 PCI_EN、MCBSP2_EN、HPI 数据引脚 HD5 和 EMIFB 地址总线（BEA[20:1]）决定。详情请查看具体的芯片数据手册。

TMS320C6x 系列 DSP 芯片的 BOOT 模式主要包括以下三种。

1．无BOOT模式

选择此模式，复位结束后，CPU 将从存储区地址 0 开始执行。在这种模式下，必须确保 CPU 从一开始就能执行有效指令，以避免得到意外的结果。

2．非可变存储区（ROM/Flash）BOOT

选择此模式，外设通过 DMA/EDMA 将代码从外部的非可变存储区传送到 DSP 存储区地址 0 开始的存储器空间内，在此期间，CPU 内部保持在复位状态。传送完毕后，DSP 从地址 0 开始执行代码。

对于 TMS320C6x 系列 DSP，ROM 的 BOOT 差别不大：

C620x/CC670x：DMA 用默认的 ROM 时序从 CE1 的开始复制 64K 字节数据到地址 0 开始的 DSP 存储器空间。

C621x/CC671x/CC64x：DMA 用默认的 ROM 时序从 CE1 的开始复制 1K 字节数据到地址 0 开始的 DSP 存储器空间。一些 C64x 芯片有多个 EMIF，请参考各自的数据手册以决定哪个 EMIF 用于 BOOT。

C62x/C67x 芯片允许选择 ROM 的宽度，它们的 EMIF 口自动将 8-bit 的字节或 16-bit 的半字连续地装配成 32-bit 的指令字来传送。C64x 芯片只支持 8-bit 的 ROM BOOT。

在开发阶段，DSP 芯片外部连接的 ROM/Flash 一般不会写入信息，但这种做法可能会引起一些潜在的问题。比较典型的是，当外部接的是一块未烧写的 Flash 芯片时，读入的代码全部为 0xFFFFFFFF，这解码为 NOP 指令。这个指令本身没有问题，但 DSP 将持续执行，直到超出所定义的存储空间，从而带来麻烦，甚至会导致仿真器无法正常工作。为避免此类问题，在开发阶段最好在 Flash 芯片中烧写一小段代码，以确保仿真器能正常工作，如以下的一段代码例子。

```
Infinite_loop:  B    .S1   Infinite_loop
                NOP
                NOP
                NOP
                NOP
                NOP
                NOP
                NOP
```

3．主机BOOT

选择此模式，在 BOOT 期间 CPU 将保持内部的复位状态。此时外部主机可以测试主机接口（HPI，PCI 或 XBUS），进入 DSP 的内部或外部存储区，初始化 DSP 的内部配置寄存器。在所有这些必要的工作结束后，主机设置 DSPINT 以结束 BOOT 进程，唤醒 DSP 从地址 0 开始执行。这个中断不会导致 DSP 执行中断服务进程，因为此前 DSP 处于复位状态。但 CPU 必须清除 DSPINT 以便主机后续的中断的执行。若系统设计为用主机控制 BOOT 模式，需确保在复位结束时，BOOT 模式选择脚处于正确的状态。

在早期的 C6x 芯片中，如果选择了主机 BOOT 模式，DSP 将等待主机初始化 DSP 并设置 DSPINT。若主机依赖 DSP 的复位信号开始传送代码，那么主机可能会随机停止，因为在仿真器复位期间无法看到复位信号变高。

10.4 小　　结

本章讨论了 DSP 脱机系统的设计问题。首先以 TMS320C54x 为例介绍了 DSP 芯片的初始化，接着介绍了 TMSS320C50、TMS320C54x、TMS320C55x 和 TMS320C6x 系列芯片的脱机系统设计问题。

在脱机系统设计中，最关键的问题有以下几点：（1）DSP 芯片复位后，如何初始化配置 CPU 及外设寄存器？（2）DSP 芯片如何进行 BOOT 方式的选择？（3）在 DSP 芯片处于 BOOT 的主导方情况下，DSP 芯片通过哪种接口，从什么地址开始、将多少数据搬移到什么位置的片内程序存储区？对接口的硬件设计有什么要求？（4）在 DSP 芯片处于 BOOT 的从方情况

下，外部主机通过什么接口，将执行代码写入到什么位置的片内存储区？DSP 芯片如何得知 BOOT 进程结束，从而开始执行搬移入的用户代码？（5）DSP 脱机系统在 BOOT 进程结束后，从什么地址开始执行代码？地址是如何得到的？

把握了以上几点，那么设计和实现 DSP 脱机系统将不会对读者造成很大的困难。本章对有关芯片的 BOOT 设计仅作了简要介绍，如果需要了解更详细的内容，读者可查看相应芯片的用户手册。

10.5　习题与思考题

（1）编写一个对 TMS320C54x 进行初始化的程序，要求：

a）矢量表的首地址为 7E00h；

b）时钟模式是 PLL×5；

c）FRCT=1；

d）I/O 口访问时自动插入一个等待状态。

（2）对 TMS320VC5402 的 McBSP0 串行口进行初始化配置，要求 16 位字长，内部提供串行口时钟（串行时钟 2 MHz、同步时钟 8 kHz），DSP 芯片的工作时钟为 100 MHz。

（3）以 TMS320VC5402 为例，设计实现外接 8 位 EPROM 进行程序引导。

（4）在 TMS320C5509A 脱机系统中，EHPI　BOOT 方式要求最后写入 00 0060h 地址的内容，为什么？试以 TMS320C5402 为主机，对 TMS320C5509A 的 EHPI　BOOT 方式进行软硬件的设计。

第 11 章　DSP 硬件系统设计

11.1　引　　言

DSP 应用系统包括硬件系统和软件系统两部分，硬件系统设计是实现整个应用系统的基础，本章重点介绍 DSP 硬件系统的设计方法。

根据系统功能划分，DSP 硬件系统设计可以分为最小硬件系统设计和外围硬件接口设计。最小硬件系统只包含了 DSP 的基本电路模块，在最小硬件系统的基础上利用 DSP 芯片的外设接口进行扩展，可以实现更为复杂的系统。本章首先介绍 DSP 最小硬件系统中基本的硬件设计和常用的可靠性设计方法，最后讨论典型的硬件接口设计方法。

11.2　DSP 系统的基本硬件设计

DSP 基本硬件系统包括复位电路、电源电路、时钟电路、JTAG 电路以及其他基本外设。其中，常用片内外设电路的设计已在第 9 章进行了介绍。虽然根据基本的电路连接原理就可以实现完整的电路系统，但是由于外围电路器件的种类非常多，针对不同的应用场合有不同的指标要求，片内外设要和外围电路连接时需要考虑电平、功耗等多方面的问题。本节将对这些问题进行分析。

11.2.1　电源电路

11.2.1.1　电源需求

超大规模集成电路从 1 μm 发展到目前的 0.1 μm，芯片的供电电压和功耗随之降低。DSP 芯片的电源从最初的 5 V 降低到目前的 3.3 V，核心电压降低到 1 V 左右。目前主流的 DSP 芯片都采用低电压供电方式，外围电压均已采用 3.3 V，并采用内核电压和 I/O 电压分开的方式。5 V 的 DSP 芯片的价格和功耗相对较高，已逐渐被 3.3 V 的 DSP 芯片所取代。

以 TMS320LC/VC549 为例子，该芯片的电源分两种：内核电源（CVdd）和 I/O 电源（DVdd），其中 I/O 电源一般采用 3.3 V 电压，TMS320LC549 的内核电压为 3.3 V，TMS320VC549 的内核电压是 2.5 V。下面以 TMS320VC549 为例介绍 DSP 芯片的电源设计。

1．电压要求

TMS320VC549 的内核工作电压为 2.5 V，主要为该芯片的内部逻辑提供电压，包括 CPU 和其他所有的外设逻辑。与 I/O 电源 3.3 V 供电相比，2.5 V 供电可以降低 44%的功耗。外部接口引脚采用可与常用低压器件相匹配的电压，与其他器件相连无需电平转换电路。

2．电流要求

TMS320VC549 的电流消耗主要取决于器件的激活度。CVdd 为器件的所有内部逻辑提供电流，包括 CPU、时钟电路和所有外设，其消耗的电流主要决定于 CPU 的激活度。外设消耗的电流决

定于正在工作的外设及其速度，一般外设消耗的电流与 CPU 相比是比较小的。时钟电路需要消耗一小部分电流，这部分电流是恒定的，与 CPU 和外设的激活程度无关。DVdd 只为外部接口引脚提供电压，消耗的电流决定于外部输出的速度和数量，以及在这些输出上的负载电容。

下面来估计一下 TMS320VC549 的工作电流。这里估计的是峰值电流，以便选择合适的电压调节器。TMS320VC549 的每 MHz（CLKOUT）消耗的平均电流是 0.45 mA，峰值电流为每 MHz（CLKOUT）1.1mA。这里按每 MHz（CLKOUT）1.5 mA 来计算，TMS320VC549 工作在 100 MHz 的最大估计电流为：

（1.5mA/MHz）×（100 MHz）= 150 mA

3．加电次序

由于芯片采用双电源供电，因此需要考虑加电次序问题。理想情况下，DSP 芯片上的两个电源同时加电，但是在一些场合很难满足这一要求。如果不能做到同时加电，应先对 DVdd 加电，然后对 CVdd 加电。DVdd 应不超过 CVdd 电压 2 V。这个加电次序主要依赖于 TMS320VC549 的内部静电放电保护电路，内部保护电路的示意图如图 11-1 所示。从图中可以看出，DVdd 不能超过 CVdd 的电压为 4 个二极管压降（约 2 V），CVdd 不能超过 DVdd 的电压为 1 个二极管压降（约 0.5 V），否则有可能损坏芯片。

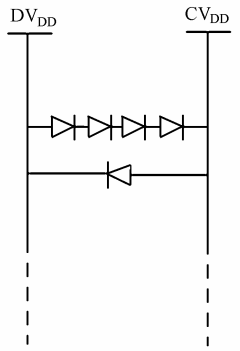


图 11-1　TMS320VC549 内部保护电路

11.2.1.2　双电源解决方案

采用什么供电机制主要取决于应用系统中提供什么样的电源。考虑到大部分数字系统工作于 5 V 和 3.3 V，下面我们来考虑两种情况。

（1）从 5 V 产生：图 11-2 示出了从 5 V 生成 3.3 V 和 2.5 V 的一种方案。其中，第 1 个电压调节器（Regulator）提供 3.3 V，而第 2 个电压调节器提供 2.5 V。

（2）从 3.3 V 产生：图 11-3 示出了从 3.3 V 生成 2.5 V 的一种方案。其中，电压调节器提供 2.5 V 电压。

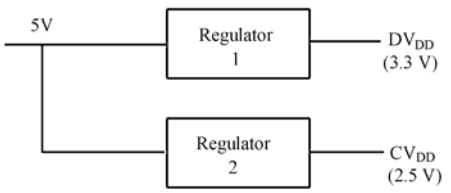


图 11-2　从 5 V 产生双电源

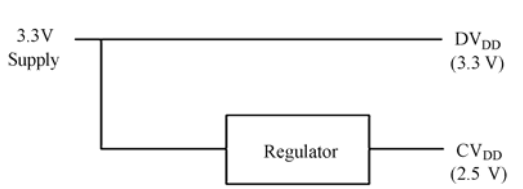


图 11-3　从 3.3 V 产生双电源

各个大型芯片公司都提供了系列的电源芯片，根据上面不同的供电机制，我们可以合理选择系统实现时的电源芯片。例如，可以采用如下三种解决方案：

（1）单 3.3 V 电压输出。单 3.3 V 电压输出可以选用 TI 公司的 TPS7133、TPS7233、TPS7333，或其他公司的芯片，如 Maxim 的 Max604 等。

（2）单电源可调电压输出。TI 公司的 TPS7101、TPS7201 等芯片提供可调节的输出电压。电压调节通过改变外接的两个电阻的阻值实现。

（3）双电源输出。TI 公司也提供了有两路输出的电源芯片，如 TPS73HD301、TPS73HD325、TPS73HD318。其中，TPS73HD301 的输出电压为一路 3.3 V、一路可调输出，TPS73HD325 的输出电压为一路 3.3 V、一路 2.5 V，TPS73HD318 的输出电压为一路 3.3 V、一路 1.8 V。每路电源的最大输出电流为 750 mA。芯片还提供两个宽度为 200 ms 的低电平复位脉冲。图 11-4 为 TPS73HD325 的一种应用电路。

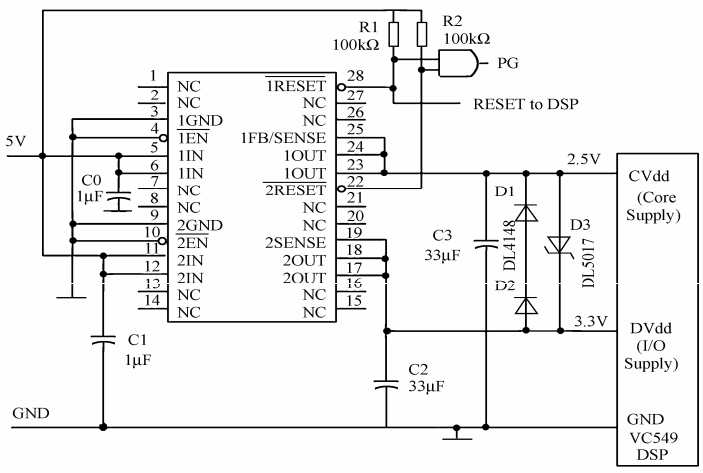


图 11-4　TPS73HD325 双电源 DSP 应用电路

11.2.2　复位电路

DSP 芯片上有一个复位引脚 $\overline{\text{RS}}$，用于外部对芯片的复位。一般来说，复位信号都是低电平有效。图 11-5 示出了一个简单实用的上电复位电路。电源刚加上时，DSP 芯片处于复位状态。为使芯片初始化正确，一般应保证 $\overline{\text{RS}}$ 有效期间至少持续 3 个 CLKOUT 周期。但是，在上电后，晶体振荡器需要一定时间的稳定期，一般为 100~200 ms。图 11-5 的复位时间主要由 R 和 C 确定。A 点的电压 $V = V_{CC}\left(1-e^{-t/\tau}\right), \tau = RC$。设 V_1＝1.5 V 为低电平与高电平的分界点，则：

$$t_1 = -RC\ln\left[1-\frac{V_1}{V_{CC}}\right]$$

选择 R＝100 kΩ，C＝4.7 μF，可得 t_1＝167 ms，随后的施密特触发器保证了低电平的持续时间至少为 167 ms，从而满足复位要求。

实际上由于 DSP 系统的时钟频率较高，在运行时极有可能发生干扰和被干扰的现象，严重时系统可能会出现死机现象。为了克服这种情况，除了在软件上作一些保护措施外，硬件上也需要作相应的处理。硬件上最有效的保护措施就是采用具有监视（Watchdog）功能的自动复

位电路，自动复位电路除了具有上电复位功能外，还具有监视系统运行，并在系统发生故障或死机时再次进行复位的能力。其基本原理就是通过电路提供一个用于监视系统运行的监视线，当系统正常运行时，应在规定的时间内给监视线提供一个高低电平发生变化的信号，如果在规定的时间内这个信号不发生变化，自动复位电路就认为系统运行不正常并重新对系统进行复位。

根据上述原理，可以用常用的器件（如 555 加上一些计数器）设计自动复位电路。此外也有许多专门的自动复位电路，如 MAX 系列。图 11-6 是采用 MAX706 实现的自动复位电路，其引脚 6 输入的是系统提供的监视信号，引脚 7 输出的是复位信号，其低电平复位时长约 200 ms。

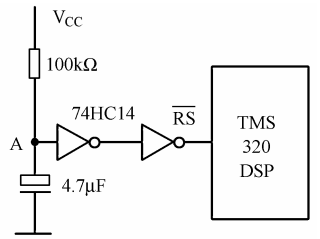


图 11-5　简单的复位电路

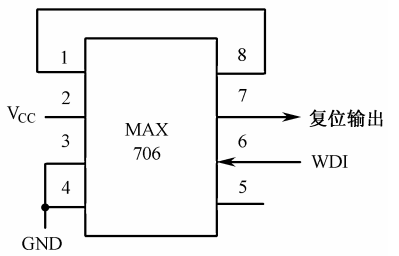


图 11-6　具有 Watchdog 功能的复位电路

11.2.3　引脚的电平转换与处理

11.2.3.1　电平转换

TMS320C54x 等新一代 DSP 芯片的 I/O 工作电压是 3.3 V，因此，其 I/O 电平也是 3.3 V 逻辑电平。在设计 DSP 芯片与其他外围芯片的接口时，如果外围芯片的工作电压也是 3.3 V，那么直接连接就可以了。但是，目前还有很多外围芯片采用 5 V 接口电平，如 EPROM、SRAM、模数转换芯片等，因此，就存在一个如何将 3.3 V DSP 芯片与这些 5 V 供电芯片的可靠接口的问题。

1．各种电平的转换标准

图 11-7 示出了 5 V CMOS、5 V TTL 和 3.3 V TTL 电平的转换标准。其中，V_{OH} 表示输出高电平的最低电压，V_{IH} 表示输入高电平的最低电压，V_{IL} 表示输入低电平的最高电压，V_{OL} 表示输出低电平的最高电压。从图中可以看出，5 V TTL 和 3.3 V TTL 的转换标准是一样的，而 5 V CMOS 的转换电平是不同的。因此，在将 3.3 V 系统与 5 V 系统接口时，必须考虑到两者的不同。

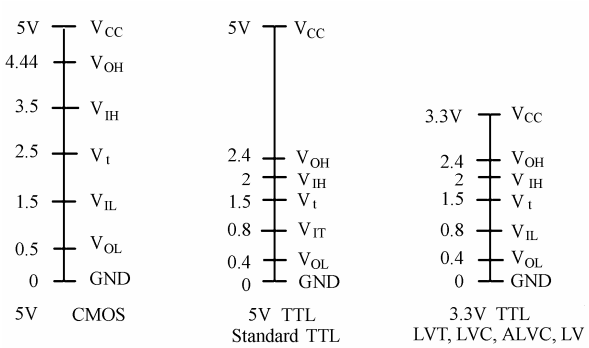


图 11-7　5 V CMOS、5 V TTL 和 3.3 V TTL 电平转换标准

2．3.3 V与 5 V电平转换的四种情形

根据实际应用场合，下面我们来考虑 4 种不同的情况，如图 11-8 所示。

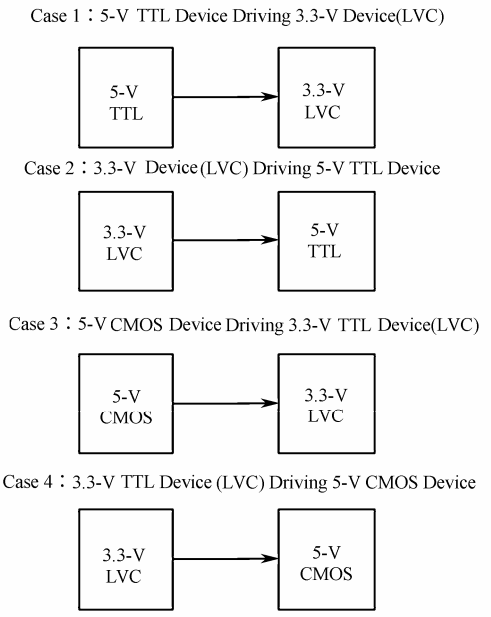


图 11-8　3.3 V 器件与 5 V 器件接口时的 4 种情况

（1）5 V TTL 器件驱动 3.3 V TTL 器件。由于 5 V TTL 和 3.3 V TTL 的电平转换标准是一样的，因此，如果 3.3 V 的器件能够承受 5 V 电压，直接相接从电平上来说是完全可以的。

（2）3.3V TTL 器件（LVC）驱动 5V TTL 器件。由于两者的电平转换标准是一样的，因此，不需要额外的器件就可以将两者直接相接。不需要额外的电路直接从 3.3 V 器件驱动 5 V 器件看起来不可思议，但只要 3.3V 器件的 V_{OH} 和 V_{OL} 电平分别是 2.4 V 和 0.4 V，5 V 器件就可以将输入读为有效电平，因为它的 V_{IH} 和 V_{IL} 电平分别是 2 V 和 0.8 V。

（3）5V CMOS 驱动 3.3V TTL 器件（LVC）。显然，两者的转换电平是不一样的。进一步分析 5V CMOS 的 V_{OH} 和 V_{OL} 以及 3.3V LVC 的 V_{IH} 和 V_{IL} 的转换电平可以看出，虽然两者存在着一定的差别，但是能够承受 5 V 电压的 3.3 V 器件能够正确识别 5 V 器件送来的电平值。采用能够承受 5 V 电压的 LVC 器件，5 V 器件的输出是可以直接与 3.3 V 器件的输入端接口的。

（4）3.3V TTL 器件（LVC）驱动 5 V CMOS。两者的电平转换标准是不一样的，从图 11-7 可以看出，3.3V LVC 的 V_{OH} 最低电压值是 2.4 V（可以高到 3.3 V），而 5 V CMOS 器件要求的高电平最低电压值是 3.5 V，因此，3.3 V 器件（LVC）的输出不能直接与 5 V CMOS 器件的输入相接。在这种情况下，可以采用双电压（一边是 3.3 V 供电，另一边是 5 V 供电）供电的驱动器，如 TI 的 SN74ALVC164245、SN74LVC4245 等。

3．DSP与外围器件的接口方法

下面我们考虑两种情形。

（1）与 3 V 器件的接口：与 3 V 器件（如 3 V Flash）接口比较简单，由于两者电平一致，可以直接相接。

（2）与 5 V 器件的接口：首先需要仔细分析 5 V 器件的电平转换标准，这可以从器件的电

气性能说明中获得。一般来说 DSP 输出给 5 V 的电路（如 D/A），无需加任何缓冲电路，可以直接连接。DSP 输入 5 V 的信号（如 A/D），由于输入信号的电压>4 V，超过了 DSP 的电源电压，DSP 的外部信号没有保护电路，需要加缓冲，如 74LVC245 等，将 5 V 信号变换成 3.3 V 的信号。

下面以 TMS320LC549 与 Am27C010 EPROM 接口为例来说明接口的设计方法。首先分析 Am27C010 的电平转换标准。从 TMS320LC549 和 Am27C010 的电气性能说明可知它们的电平转换标准，如表 11-1 所示。

表 11-1　电平转换标准

器件	V_{OH}	V_{OL}	V_{IH}	V_{IL}
C549	2.4 V	0.4 V	2.0 V	0.8 V
Am27C010	2.4 V	0.45 V	2.0 V	0.8 V

从表中可以看出，C549 与 Am27C010 的电平转换标准是一致的，因此，从 C549 单向到 Am27C010 的地址线和信号线可以直接与之相接。但由于 C549 不能承受 5 V 电压，因此从 Am27C010 到 C549 的数据线不能直接相接。解决的办法就是在中间增加一个缓冲器件，这个缓冲器件可以是双电压供电，也可以是能承受 5 V 电压的 3.3 V 单电压供电器件。我们选择 74LVC16245 作为缓冲器件来设计两者的接口。

先来看一下 74LVC16245 的基本功能。74LVC16245 是工作电压为 2.7~3.6 V 的双向收发器，可以用作 2 个 8 位收发器或 1 个 16 位收发器，根据方向控制端（DIR）的电平可允许数据从 A 端传输到 B 端，或从 B 端传输到 A 端。输出使能控制线（$\overline{OE}$）可用来使器件有效或无效（双侧相互隔离）。操作方式如表 11-2 所示。

表 11-2　74LVC16245 的操作方式

$\overline{OE}$	DIR	操作
L	L	B→A
L	H	A→B
H	X	隔离

Am27C010 是 EPROM，其数据只是从 EPROM 向 DSP 单向传输。DSP 与 Am27C010 的接口示意图如下图 11-9 所示。

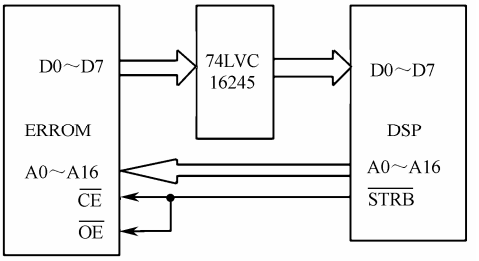


图 11-9　DSP 与 5V EPROM 的接口

除了使用上面介绍的 74LVTH16245 这类总线收发器（Bus Transceiver）进行电平转换外，还可以使用如下一些方法：

（1）总线开关（Bus Switcher）：常用的器件有 SN74CBTD3384（10 位）、SN74CBTD16210（20 位）等，这类器件 5 V 供电，无需方向控制，可以用于信号方向灵活、且负载单一的应用，

如 McBSP 等外设信号的电平变换。

（2）2 选 1 切换器（1 of 2 Multiplexer）：常用的器件有 SN74CBT3257（4 位）、SN74CBT16292（12 位）等，这类器件 5 V 供电，无需方向控制，可以实现 2 选 1，适用于多路切换信号、且要进行电平变换的应用，如双路复用的 McBSP 等情况。

（3）CPLD ：CPLD 一般都用 3.3 V 供电，但其引脚的输入容限为 5 V，可以用于增强输出驱动的情况。但这种方法延迟较大（通常＞7 ns），因此只适用于少量的对延迟要求不高的信号。

（4）电阻分压：利用大电阻串连的方法进行分压，这种方法由于增加了电流损耗，因此一般只适用于调试过程中或非产品化系统中。

另外还要注意的是仿真器的 JTAG 口的信号也必须为 3.3 V，否则有可能损坏 DSP。

11.2.3.2 未用引脚的处理

DSP 芯片上有很多引脚，在很多系统中有一些引脚没有使用。这些空闲引脚根据其性质不同，需要进行不同的处理。

1．未用的输入引脚

未用的输入引脚不能悬空不接，否则容易在这些影响上产生虚信号，影响片内的正常工作。一般应将它们上拉或下拉为固定的电平。

（1）关键的控制输入引脚：Ready、Hold 等关键引脚应固定接为适当的状态。Ready 引脚应固定接为有效状态，Hold 引脚应固定接为无效状态。

（2）无连接（NC）和保留（RSV）引脚：NC 引脚除非特殊说明外，需要悬空不接；RSV 引脚应根据芯片手册具体决定接还是不接。

（3）非关键的输入引脚：这些引脚可以上拉或下拉为固定的电平。

2．未用的输出引脚

未用的输出引脚可以悬空不接。

11.2.4 硬件系统的可靠性设计

为提高 DSP 系统的可靠性，必须考虑接地、屏蔽、隔离等技术的应用。同时，由于 DSP 的工作频率较高，芯片在运行过程中会产生较大的功耗，如果处理不当也将影响系统的可靠性。本节将针对这些问题进行分析。

11.2.4.1 接地

1．接地方式

常用的接地方式有两种：安全接地和工作接地。

为了设备和人身的安全，将设备的外壳接地，这种方式称为安全接地（或称为外壳接地）。这种方式是真正的接地。安全接地时，接地电阻要尽可能低，在材料等方面都会有一定的要求。

工作接地：为了电路工作需要，将零电位参考点相连，这种方式称为工作接地。在很多情况下，工作的零电位相对地球的大地是浮空的，因此这种工作地是“悬浮地”。一般的电子系统或便携系统都是工作接地方式。

2．接地系统的实现

对于一个较大的电子系统，应根据信号的指标和电源类别等分类接地，实现完整的接地系统。通常需要将如下三类器件分别处理：

（1）弱信号地：将系统中的小信号回路、控制回路、逻辑电路以及直流电源等连在一起接地。这时弱信号地是工作地。

（2）功率地：系统中常常可能包含有一些大功率器件，例如继电器以及驱动电路等，这些器件连在一起构成功率地。由于功率较大，这些电路可能会成为干扰弱信号回路的噪声源，因此功率地与弱信号地不可混接。

（3）机壳地：系统的外壳等金属构件接地，这通常是所谓的安全地。

这几种接地系统可以相互独立，也可以成伞状接在一起（如图 11-10 和 11-11 所示），需要注意的是，不能将功率地和弱信号地接在一起后再与机壳地相连，这样有可能因为存在功率电路和弱信号回路间的阻抗而产生噪声，影响弱电回路。

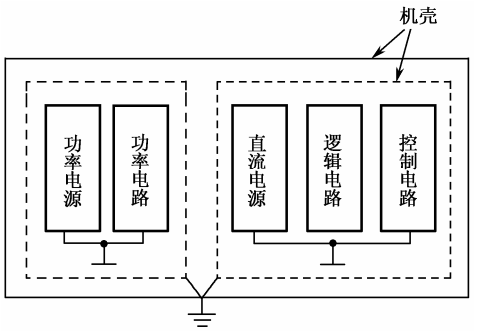


图 11-10　独立的系统接地

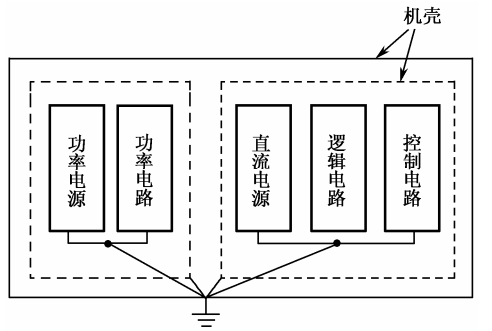


图 11-11　伞状系统接地

11.2.4.2　屏蔽

高频电源、交流电源、强电设备产生的电火花，以及雷电等自然现象，都能产生电磁波，从而成为电磁干扰的噪声源。当距离较近时，电磁波会通过分布电容和电感耦合到信号回路形成电磁干扰，当距离较远时，电磁波会以辐射形式构成干扰。在利用 DSP 实现高频信号处理时经常会遇到这样的情况，另外，DSP 本身的振荡器由于频率较高，也可能产生电磁干扰，同时也极易受其他电磁干扰的影响而干扰芯片的正常工作。

利用屏蔽技术可以克服电磁波干扰的影响。屏蔽体通常可以采用金属板、金属网或金属盒构成，它以反射方式或吸收方式来削弱电磁波，从而形成对电磁波的屏蔽作用。

高频干扰和低频干扰的屏蔽方法有所区别。低频干扰可以利用高导磁材料制作的屏蔽体，使电磁波经过屏蔽体壁的低磁阻磁路通过，而不影响屏蔽体内的电路。而屏蔽高频电磁干扰时，可以选择铜、铝等导电率高的材料。

屏蔽体也存在接地问题。为了消除屏蔽体和内部电路之间的寄生电容，应按照“一点接地”的原则设计屏蔽体的接地电路。

11.2.4.3　隔离

隔离通常可以采用物理隔离和光电隔离两种。

物理隔离是指对小信号低电平的隔离，此时信号连线应尽量远离高电平大功率的导线，以

减少噪声和电磁场的干扰。为了实行物理隔离，在同一设备的内部也应当把这两类信号的导线分开走线。同时，要注意在远距离走线时，将信号电缆和功率电缆分开，并保持一段距离，必要时可以用金属管将它们分别套起来进行保护。

光电隔离一般通过光电耦合器实现，光电耦合器包括一个发光二极管和一个光敏三极管。耦合器的外壳有金属的或塑料的两种，发光二极管和光敏三极管之间用透明绝缘体填充。输入信号可以使发光二极管发光，其光线又使光敏三极管产生电信号输出。光电耦合的响应时间一般不超过几个微秒，否则将影响其在高速电路上的应用。光电耦合器的示意图如图 11-12 所示。

由于光电耦合器的输入和输出在电气上是绝缘的，且输出端和输入端之间没有反馈，因此光电耦合器既可以隔离两端电路的电信号，实现电平转换，又可以实现抗干扰。

此外，为提高接口的性能，经常需要在电路中采用滤波技术抑制噪声干扰。例如，如果电路中既存在模拟电路又存在数字电路，那么数字电路中状态的变化会在模拟的电源上产生一个尖峰电流，从而形成数字噪声。利用电容、电感等储能元件可以抑制这样的噪声，如图 11-13 就是常用的一种滤波电路。通过在电源线的入端并联两个电容，其中 50μF 的电解电容用于抑制电源噪声的低频分量，而 0.01μF 电容抑制高频分量。如果可以在电容前面的电源线上串连一个电感，滤波效果将会更好。

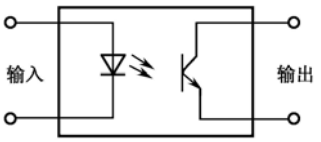


图 11-12　光电耦合器符号

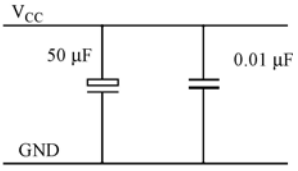


图 11-13　电源滤波器示意图

11.2.4.4　散热

1．芯片功耗

DSP 芯片由于工作频率较高，在工作时的功耗较低频器件更大一些，这是系统设计中需要考虑的问题。总的而言，TI 公司的 C5000 和 C6000 两个系列的 DSP 中，C6000 系列 DSP 主要应用于高端系统和大型系统中，功耗较大，而 C5000 系列 DSP 主要应用于便携系统和低成本系统中，因此功耗得到了控制，尤其是 C55x 系列 DSP 芯片，和 C54x 芯片相比，每 MIPS 的功耗下降了 5/6。在 C6000 系列芯片中，新推出的芯片在功耗设计上也作了很大改进，目前 C6201 仅在 2W 左右。

DSP 芯片的实际功耗会随芯片的工作状态而变，芯片的工作状态决定于具体的应用程序，包括 CPU 占用程度等。为评估功耗，一般可以定义芯片的两种活动程度：High DSP activity 和 Low DSP activity，如表 11-3 所示。

表 11-3　DSP 活动程度定义

DSP 活动程度	CPU	程序存储器 访问率	数据存储器 访问率	外部存储器 100MHz I/O
High DSP activity	8 条指令	100%	100%CPU，50%DMA	100%访问
Low DSP activity	2 条指令	25%	25%CPU	0%访问

以 C6000 系列芯片为例，影响其功耗的主要因素有如下几个方面：

- 工作频率；

- Power-down 或者 IDLE 状态；
- 片内活动模块的数量；
- 每周期运行的功能单元数；
- 片内数据单元的存取率；
- 片内程序单元的访问率；
- 片内外设（DMA、串口等）的运行情况；
- 芯片的数据吞吐速度。

2．散热设计

对于 C6000 系列芯片来说，其高集成度、高时钟频率和较大的片内存储阵列设计，使得它与低端 DSP 芯片相比，具有更大的结温系数。为了保证器件稳定可靠地工作，芯片的结温不能超过某个最大的限度。因此，系统设计中应当考虑加入合适的散热系统，协助芯片通过外封装进行散热。常用的散热方法有无源散热片、风扇、水冷系统等，其中无源散热片是一种性价比较高的有效解决方案。

在选择散热片的型号时，需要考虑：

- 电路板所在工作环境中的空气流速；
- 环境温度；
- 芯片封锁与散热片的结合方式；
- 电路板的设计与布线。

11.3　DSP 典型硬件系统举例

利用 DSP 的片内外设，可以设计出功能多样的应用系统。由于每种接口都可以采用不同的方式实现，为方便阅读，本节将按照接口的功能，选取存储器、数模/模数转换、通信等典型接口分别进行介绍。

11.3.1　外部存储器接口设计

目前 DSP 芯片的片内存储器 RAM 越来越大，要设计高效的 DSP 系统，就应该选择片内 RAM 较大的 DSP。片内 RAM 同片外存储器相比，有以下优点：

（1）片内 RAM 的速度快，可以保证 DSP 无等待运行；

（2）对于 C2000/C3x/C5000 系列，部分片内存储器可以在一个指令周期内访问两次，使得指令可以更加高效；

（3）片内 RAM 运行稳定，不受外部的干扰影响，也不会干扰外部；

（4）DSP 片内多总线，在访问片内 RAM 时，不会影响其他总线的访问，效率较高。

然而，当 DSP 的程序较大、处理的数据非常多，或者存在与外界其他芯片有大量的数据交互时，片内的 RAM 存储空间无法满足系统要求。这时需要对 RAM 存储器进行扩展。另外，对于存储程序和固定数据的 ROM 类存储器而言，虽然 DSP 片上带有一定容量的 ROM，但要利用 DSP 片内的 ROM，需要由用户将程序代码提交芯片公司进行掩膜才能实现。由于这种方法成本巨大，对于小规模的系统开发并不合适。因此一般外部存储器接口设计对 ROM 和 RAM 的扩展都需要考虑。

设计存储器接口时主要考虑存储器的接口类型和速度，以确定连线方式和时序控制。存储

器的接口类型可以分为并行和串行两种。本书只讨论并行存储器的设计方法。

11.3.1.1 TMS320C54x 存储器接口设计

1．内部RAM的使用

C54x 系列 DSP 芯片设计有丰富的内部快速存储器，使用内部存储器可以全速运行，达到芯片最高速度。因此，充分利用内部存储器可以使 DSP 系统的整体性能达到最佳。在该系列的DSP芯片中，不同的芯片具有不同数量的内部RAM，但特性是类似的。下面以TMS320LC549为例来介绍内部 RAM 的特性和使用方法。

TMS320LC549 内部具有 16 K 字的 ROM 和 32 K 字的 RAM。内部 RAM 和 ROM 可根据 PMST 寄存器中的 OVLY、DROM 位灵活设置。对于一般应用，应尽量采用内部 RAM，因此应作如下设置：

（1）将芯片的 MP/$\overline{\text{MC}}$接低，使芯片工作在微计算机方式。

（2）设置 OVLY=1，使片内的 80H~7FFFH 既映射在程序区，又映射在数据区。

（3）如果要在数据区访问片内的 ROM 区，应设置 DROM=1。

片内存储空间的程序区和数据区可以重叠，在编程时需要注意。

2．外部RAM接口的设计

除了内部 32 K RAM 和 16 K ROM 之外，TMS320LC549 还可以扩展外部存储器。其中，数据空间总共为 64 K（0000H~FFFFH）, I/O 空间为 64 K（0000H~FFFFH），程序空间为 8 M。8 M 程序空间的寻址是通过额外的 7 根地址线（A16~A22）实现的，由 XPC 寄存器控制。

C54x 系列 DSP 芯片分别利用独立的选通信号$\overline{\text{PS}}$和$\overline{\text{DS}}$线用于选择程序和数据空间，一般的做法是分别用$\overline{\text{PS}}$和$\overline{\text{DS}}$选择两个不同的地址空间。ROM 一般都映射在程序存储区，连接方法较为简单，而 RAM 既可以用作程序存储器，也可用作数据存储器，因此其电路连接较为复杂一些。为避免使用多个 RAM 分别作为程序存储器和数据存储器使用，可以利用程序和数据共用存储器，方法是将$\overline{\text{PS}}$和$\overline{\text{DS}}$信号线接至一与非门形成 PDS 信号，这个信号不论是$\overline{\text{PS}}$有效（低电平）还是$\overline{\text{DS}}$有效（低电平）都呈现有效（高电平），将这个信号经反向用作片选信号就可保证将 RAM 同时用作程序和数据空间，如图 11-14 所示。

图 11-14 中 A15 与 PDS 接至与非门，从而保证了$\overline{\text{PCS_RAM}}$信号只有当 A15 和 PDS 同时为高电平时才变为有效的低电平，所以$\overline{\text{PCS_RAM}}$的寻址空间是 8000H~FFFFH。图示的电路使得不论是程序还是数据都可访问 8000H~FFFFH 中的任一地址。

为了保证系统的正确运行，一般需将这 32K 空间划分为程序区和数据区，如程序占据 8000H~BFFFH 前 16 K，数据占据 C000H~FFFFH 的后 16 K，也可以是程序占 8 K，数据占 24 K 等，划分完全取决于应用程序的需要。程序员可根据实际系统的情况灵活地划分程序和数据空间，但不管如何划分，必须保证程序和数据区的相互分离，以免形成冲突。下面介绍几种扩展分离的方法。

（1）外接一个 128 K×16 的 RAM，程序区和数据区分开

图 11-15 为采用 128 K RAM 实现分开程序区和数据区的接口方法，采用程序选通线（/PS）接外部 RAM 的 A16 地址线实现。因此，程序区为 RAM 的前 64 K（0000H~FFFFH），数据区为 RAM 的后 64 K（10000~1FFFFH）。对 DSP 而言，程序区和数据区的地址均为 0000H~FFFFH。

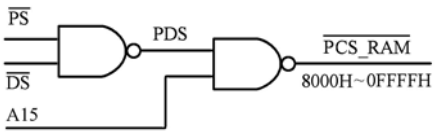


图 11-14　程序和数据共用 RAM

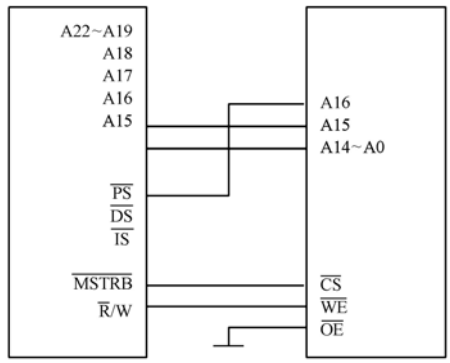


图 11-15　分开的程序和数据空间配置

采用这种外部存储器配置，需要注意以下几点：

- 如果内部 RAM 设置为有效，则相同地址的外部 RAM 自动无效。
- 当外部 RAM 的存取速度不能全速运行时，需要根据速度设置插入等待状态（设置 SWWSR 寄存器）。

（2）混合程序区和数据区

当 OVLY=1 时，内部 RAM 既是数据区又是程序区，这样设置的好处是程序可以在内部全速运行。缺点是由于程序和数据是共用的，存储区就变小了，此外，在链接时必须将程序和数据分开，以避免重叠。

外部存储器也可以采用这种方法。不采用$\overline{PS}$和$\overline{DS}$选通线，直接采用$\overline{MSTRB}$接至 RAM 的片选端，这样外部 RAM 既作为程序区也作为数据区。当然，需要注意的是，如果 DSP 外部还外接其他外设，则必须注意地址的重叠。此外，由于数据区的寻址范围只能是 0000H~FFFFH，因此，SRAM 的高 64K 地址不能被寻址到。如图 11-16 所示。

（3）一种优化的混合程序和数据区外接 RAM 方法

图 11-17 示出了一种优化的混合程序和数据区外接 RAM 方法。这种方法省去了 DSP 的 A15 地址线，将 RAM 分为 32K 长度的块。采用这种方法，可充分利用外接的 RAM，不会因内部 RAM 和外部 RAM 的地址重叠而造成外部 RAM 的浪费。下面分析一下外部 RAM 的地址分配。

① 外部 RAM 的 0000H~7FFFH 对应于 DSP 的数据区的 8000H~FFFFH 和程序区的 08000H~0FFFFH 及 18000H~1FFFFH；

② 外部 RAM 的 8000H~FFFFH 对应于 DSP 程序区的 28000H~2FFFFH 和 38000H~3FFFFH；

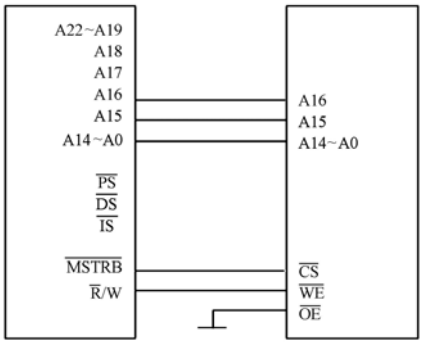


图 11-16　混合的程序和数据空间配置

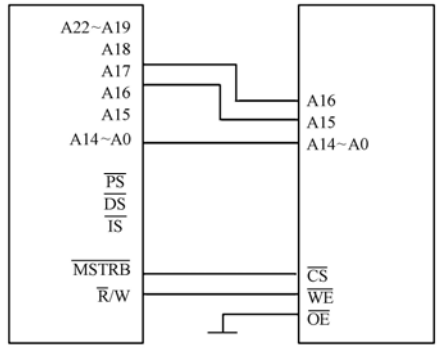


图 11-17　优化的混合程序和数据空间配置

③ 外部 RAM 的 10000H~1FFFFH 对应于 DSP 程序区的 48000H~4FFFFH 和 58000H~5FFFFH；

④ 外部 RAM 的 18000H~1FFFFH 对应于 DSP 程序区的 68000H~6FFFFH 和 78000H~7FFFFH。

当然，由于外部扩展的程序空间很大，DSP 程序区的其他地址空间也能访问到外部 RAM。这种优化的外部 RAM 配置方法，使得在使用 DSP 内部 RAM 的情况下能够充分利用外部扩展 RAM。

3．Flash 接口的设计

目前市场上的 EPROM 的工作电压一般为 5 V，与 3.3 V DSP 芯片相接时需要考虑电平转换问题，而且体积都较大。Flash 与 EPROM 相比，具有更高的性能价格比，使用比较方便，3.3 V Flash 可直接与 DSP 相接。因此，采用 Flash 作为存储程序和固定数据是一种比较好的选择。这里介绍一种 Intel 公司的 Flash 存储器（28F400B3）及其与 DSP 的接口。

（1）Flash 的控制逻辑信号

Flash 有 6 根控制逻辑信号，如表 11-4 所示。

表 11-4　Intel 28F400B3 的控制逻辑信号

信号	定义及作用
$\overline{CE}$	片选
$\overline{OE}$	输出控制
$\overline{WE}$	写控制
$\overline{RP}$	复位
$\overline{WP}$	写保护
V_{PP}	电源

在读模式，28F400B3 的读时序与典型的存储器读时序兼容（即 $\overline{CE}$ 和 $\overline{OE}$ 为逻辑低，$\overline{WE}$ 为逻辑高）。在编程/擦除模式，28F400B3 的时序与典型的存储器写时序类似（即 $\overline{CE}$ 和 $\overline{WE}$ 为逻辑低，$\overline{OE}$ 为逻辑高）。

（2）控制命令

Flash 的工作状态是由命令来控制的，这通常由主机（如 DSP 芯片）来完成。Flash 有一个控制命令集，这里我们仅给出 3 个常用的命令，如表 11-5 所示。

表 11-5　Flash 的控制命令

命令代码	工作模式	命令含义
FF	读取	从 Flash 中读数据
40	编程	将数据写入到指定地址
20	擦除	擦除 Flash 的块

Flash 存储器内部以扇区来组织，对 Flash 存储器编程是以块为单位进行的。扇区的长度与器件有关。在对 Flash 编程前，必须先对扇区进行擦除。编程和擦除都需要两个总线周期的操作，如表 11-6 所示。

表 11-6　28F400B3 编程/擦除总线周期

命令	第 1 个总线周期			第 2 个总线周期		
	操作	地址	数据	操作	地址	数据
编程	写	X	40H	写	程序地址	程序数据
擦除	写	X	20H	写	扇区地址	D0H

编程/擦除初始化后，可用读操作检查编程/擦除的状态，当编程/擦除完成后，主机应发一个读命令，使 FLASH 进入正常的读操作状态。

（3）TMS320C54x 与 Flash 的接口

图 11-18 示出了 TMS320C54x 与 Flash 的一种接口方法。

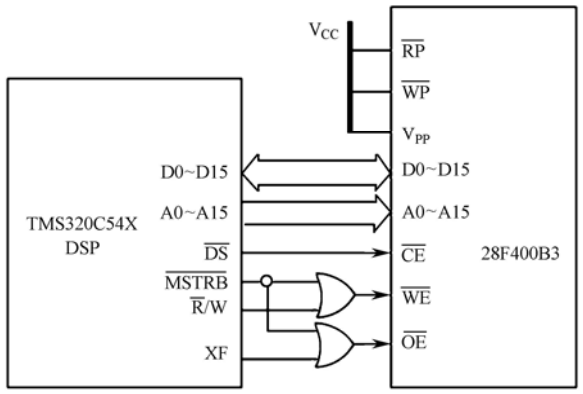


图 11-18　DSP 与 Flash 的接口

图中，28F400B3 作为 DSP 的外部数据存储器，地址总线和数据总线接至 DSP 的外部总线，$\overline{CE}$ 接至 DSP 的 $\overline{DS}$ 引脚。DSP 上的 XF 引脚用于启动编程，当 XF 为低时，Flash 处于读状态，当 XF 为高时，Flash 可擦除或编程。为了满足 28F400B3 的时序要求，XF 与 $\overline{MSTRB}$ 相或后接至 $\overline{OE}$ 。R/$\overline{W}$ 引脚与 $\overline{MSTRB}$ 相或后接至 $\overline{WE}$ 。

（4）软件设计

Flash 的编程/擦除的时序由 DSP 提供，DSP 程序在片内运行，编程流程图和擦除流程图分别如图 11-19 和图 11-20 所示。

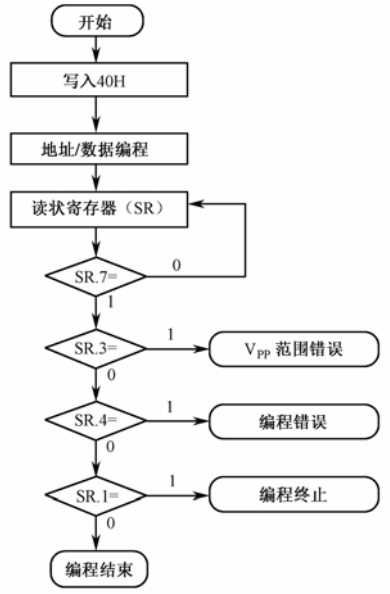


图 11-19　Flash 的编程流程图

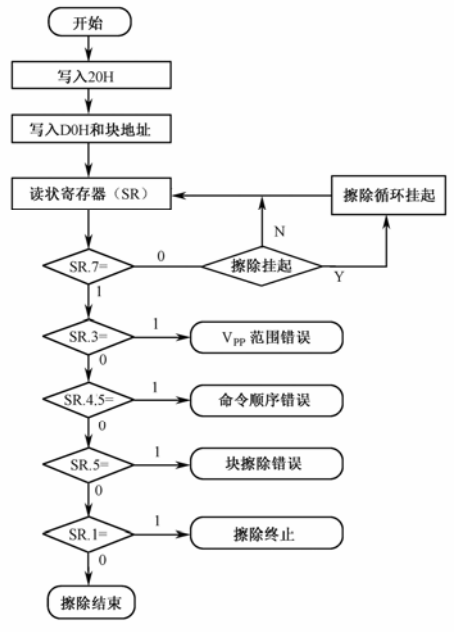


图 11-20　Flash 擦除流程图

下面是 Flash 操作的示例程序。

```
. mmregs
. def      Start
. text
Start:
          STM       #00FEH, SP
          STM       #03B4H, SWWSR     ; set s/w wait states= 5
          CALL   Set_Read_Mode        ; put into normal read mode
          ;         .
          ;         .
          STM       #0, AR1           ; ar1 is set to block address
          CALL   Erase
          STM       #4000H, AR1       ; ar1 is set to data address
          LD        #55AAH, B         ; b contains data to program
          CALL   Program
          CALL   Set_Read_Mode
          ;         .
          ;         .
Set_Read_Mode:
          SSBX   XF                   ; enable command write
          LD        #0FFH, A          ; read array command code
          STM       #4000H, AR1       ; any external memory
          STL       A, *AR1           ; write command code
          RPT       #4
          NOP                         ; delay CPU
          RSBX   XF                   ; disable command write
          RET
Erase:
          PSHM   AR1                  ; save block address
          SSBX   XF                   ; enable command write
          LD        #20H, A           ; set command code
          STM       #4000H, AR1       ; any external address
          STL       A, *AR1           ; first byte = 20h
          LD        #0D0H, A          ; second byte = 0d0h
          STL       A, *AR1
          RPT       #4
          NOP                      ; delay CPU
          RSBX   XF                ; disable command write
E_RS:
          LD        *AR1, A           ; get status
          ADD       A, #0, B          ; b=a
          AND       #80H, B           ; test sr.7
          BC        E_SC, BNEQ
          AND       #40H, A           ; erase suspend?
          BC        Error, ANEQ
          B         E_RS              ; read sr again
E_SC:
```

```
        AND     #3AH, A         ; mask error bits
        BC      Error, ANEQ
        POPM    AR1             ; restore address
        RET

Program:
        SSBX    XF              ; enable command write
        LD      #40H, A         ; set program command code
        STL     A, *AR1         ; ar1 point to external address
        RPT     #4
        NOP                     ; delay CPU
        STL     B, *AR1         ; write date
        RPT     #4
        NOP                     ; delay CPU
        RSBX    XF              ; disable command write
P_RS:
        LD      *AR1, A         ; get status
        ADD     A, #0, B        ; b=a
        AND     #80h, B         ; test sr.7
        BC      P_RS, BNEQ      ; read sr again
P_SC:
        AND     #1AH, A         ; mask error bits
        BC      Error, ANEQ
        RET
Error:
        B $                     ; self loop
.sect   ".vec"
        B   Start
.end
```

11.3.1.2　TMS320C55x 存储器接口设计

C55x 和 C6000 系列 DSP 使用 EMIF 接口访问片外存储器。第 9 章已经详细介绍了 EMIF 的控制以及对不同类型存储器的支持情况和接口设计。这里给出一个综合实例，来说明 C55x 存储器的接口设计。

在用户开发的 DSP 系统中，为降低成本，一般都将程序保存在自己选定的 ROM 器件中，而不采用掩膜的工艺。目前用户多利用 C55x 芯片实现图像处理，数字图像可以看成是由图像矩阵种的像素组成，每个像素的灰度级至少要用 6bit 表示，彩色图像一般采用 8 bit。一般分辨率的图像像素数为 256×256，高分辨率的图像像素数可达 1024×1024。这样的一幅原始图像有 1M 个像素点需要存储。而典型的 C55x 芯片 C5509 片内只有 256 K 字节的 RAM 空间，因此通常需要扩展 RAM 进行数据存储。

由于 C5509 可以扩展四个 CE 空间，并行 BOOT 方式默认的空间是 CE1。因此将 CE1 作为程序 ROM 空间进行扩展。C55x 的程序 BOOT 后从低地址开始存放，为保证程序的连续性，将最低位的扩展空间 CE0 也作为程序区，利用 RAM 进行扩展。外部数据存储区选择为 CE2，同时为方便与计算机的数据交换选择和数据的编写存储，系统中可以使用 SM 卡等存储媒质，

利用 CE3 作为 SM 卡控制器的片选信号。这样 SM 卡的访问即对应于 CE3 的寻址空间。

实现时，ROM 存储器选用 ATMEL 公司的 AM29LV800 Flash 芯片，其容量为 8 M bit。RAM 存储器选用 ISSI 公司的 IS61LV25616 SRAM，其容量为 256 K 字。由于需要分别扩展程序 RAM 和数据 RAM，因此可以使用两片 RAM。各种扩展的存储空间寻址范围为：

（1）外部程序扩展：0x02,0000~0x05,ffff （CE0 的空间范围为 0x02,0000~0x1f,ffff）；

（2）SM 卡： 由于不占用地址线，因此选择 CE3 空间地址都可以访问（CE3 的空间范围为 0x60,0000~0x7F,FFFF）。

实现的电路连接如图 11-21 所示。其中，C5509 的外部数据总线为 16 位，为简化电路连接，均采用并行 16 位的存储器件。因此 C5509 的 EMIF 字节控制引脚 BE1 和 BE0 直接与 IS61LV25616 的字节控制引脚 UB 和 LB（UB 和 LB 分别是高字节和低字节控制引脚，用于该芯片与 8 位主机相连）相连。Flash 的 BYTE 引脚直接上拉高电平，选择为字接口模式。

C5509 内部的存储空间以字节为单位寻址，由于 EMIF 接口均采用了 16 位接口器件，因此地址总线的最低位 A0 不需要使用，C5509 的地址线 A[18:1]与 RAM 的 18 位地址线相连，C5509 的地址线 A[19:1]与 Flash 的 19 位地址线相连。

除此而外，Flash 的指示引脚 RY / $\overline{\text{BY}}$ 输出到控制逻辑中，用于判断 Flash 的内部状态。为了保持 DSP CPU 对总线的有效控制，将 ARDY 接上拉电阻。

从图 11-21 中的连接可以看出，由于采用了 EMIF，多个存储器件的连接变得非常简单，不需要用户自己再进行地址编码/译码工作。同时 EMIF 接口控制也很丰富，既可与读写分离控制（两个控制引脚，$\overline{\text{WE}}$和$\overline{\text{RE}}$）的器件连接，也可与读写联合控制（仅有一个控制引脚，R/$\overline{\text{W}}$）的器件连接。

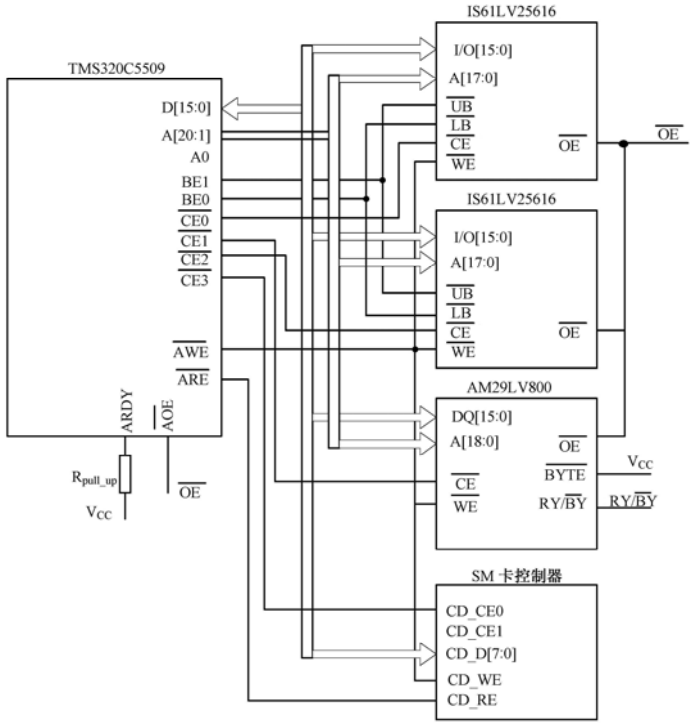


图 11-21　C5509 的 EMIF 扩展存储器电路示意图

对于动态 RAM、突发 RAM 等其他类型的 RAM 接口，这里不再举例说明。读者可以根据第 9 章的介绍，自行设计。

11.3.2　模数接口电路的设计

模数接口是 DSP 处理系统中一个重要的组成部分。目前，A/D 或 D/A 芯片在数字端采用的接口多为两种。

（1）并行的数字接口。这样的芯片与 DSP 芯片接口时需设计一定的译码电路，将转换芯片映射到 DSP 芯片的 I/O 空间，DSP 用 I/O 指令与模数接口芯片交换数据。这种芯片一般都是高速转换芯片，有的时候需要和高速 RAM 配合使用。

（2）符合一定传输协议的串行接口。由于 DSP 芯片都设计有串行接口，而且模数接口的连接电路简单，因此在低速 A/D 和 D/A 情况下，使用串行模数芯片实现 DSP 模数转换处理很多。

DSP 芯片在通信系统中的用途非常广泛，如声码器通信、调制解调器、回波抵消器等。其中使用很多集成的 A/D、D/A Codec 芯片，本节即以两种典型的 Codec 芯片为例讨论利用 DSP 的串行接口实现模数接口的设计方法。这两种芯片一种是 TI 公司生产的 TLC32044 接口芯片，另一种是 Motorola 公司的 PCM Codec 芯片 MC145483。

11.3.2.1　TLC32044 接口的设计

TLC32044 是 TI 公司生产的话带模数接口芯片（简称 AIC）TLC32040 系列中的一员。该芯片集 A/D 和 D/A 于一体，具有两个模拟输入口和一个模拟输出口，最高采样频率为 19.2 kHz，A/D 和 D/A 转换精度均为 14 位。该芯片还具有输入抗混叠开关电容滤波器和输出开关电容重构滤波器。数据输入和输出采用串行接口，数据收发有同步和异步两种工作方式，在每种方式中又可依据 WORD/$\overline{\text{BYTE}}$ 引脚的电平高低分为字工作方式和双字节工作方式两种，方便与各种 DSP 芯片的接口。TLC32044 的一个主要特点是软件可编程，工作方式、输入滤波器带宽及采样频率等参数等均可由 DSP 芯片通过软件予以设定。模数和数模变换用的基准电压可由芯片内部提供，方便了硬件设计。芯片还可以由外部控制实现自环工作，方便调试。此外，芯片还有频率微调功能，使 A/D 和 D/A 转换定时可随时调整，实现频率跟踪功能，这在 MODEM 实现中非常有用。

TLC32044 内部的各种定时信号均可以从主时钟经内部分频而得。图 11-22 是该芯片的内部定时结构框图。

从图 11-22 可以看出，AIC 在外部输出一个移位时钟 SCLK，该时钟的频率是主时钟 MCLK 的 1/4。在内部，AIC 共有四个输出频率，即接收和发送开关电容滤波器频率、A/D 转换频率和 D/A 转换频率。这四个频率可以通过改变接收或发送的寄存器 A、A’和寄存器 B 而改变。设 SCFr 和 SCFx 分别表示接收和发送的开关电容滤波器频率，Fr 和 Fx 分别表示 A/D 和 D/A 转换频率，则有

SCLK＝MCLK/4
SCFx＝MCLK/发送计数器 A
SCFr＝MCLK/接收计数器 A
Fx＝SCFx/发送计数器 B
Fr＝SCFr/接收计数器 B

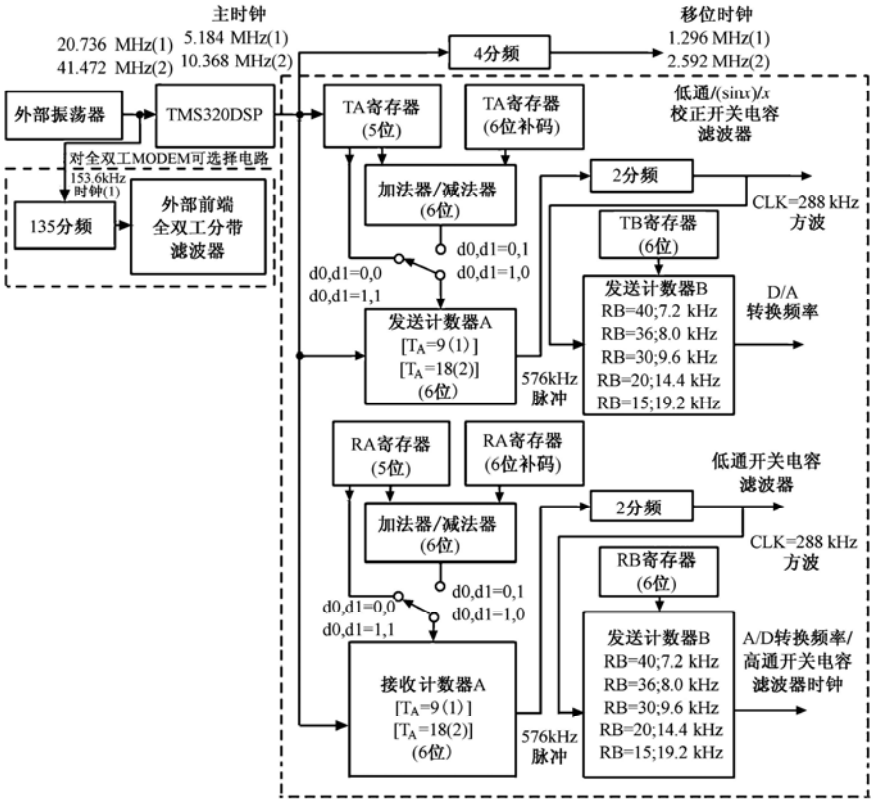


图 11-22　TLC32044 的内部定时结构

在每个 D/A 转换周期内，均可对发送计数器 A 和发送计数器 B 重新装入，同样，每个 A/D 周期内也可对接收计数器 A 和接收计数器 B 重新装入。发送和接收计数器 B 分别用 TB 和 RB 寄存器的内容装入。对发送计数器 A 的装入，可以通过软件控制有三种装入值，即 TA、TA＋TA’和 TA－TA’。采用 TA＋TA’装入，可以使即将到来的 D/A 转换滞后发生，滞后的时间量为 TA’乘以主时钟的信号周期；采用 TA－TA’装入，可以使即将到来的 D/A 转换提前发生，提前量等于 TA’乘以主时钟的的信号周期。同样，对接收计数器 A 的装入，也可有三种装入值，即 RA、RA＋RA’及 RA－RA’，其效果与 D/A 类似。

图 11-22 示出的开关电容滤波频率为 288 kHz，此时得到的输入滤波器带宽约为 3.6 kHz。在实际应用中，288 kHz 的频率很难精确得到，而且有时也需要调整输入滤波器的带宽。调整的方法是按照实际的开关电容滤波频率对 288 kHz 之比进行频率换算。例如，要得到 4.7 kHz 的输入带宽，则要求的 SCF 为

$$\frac{\mathrm{SCF}}{288\mathrm{kHz}} \times 3.6 = 4.7 \qquad => \quad \mathrm{SCF}=288\times 4.7/3.6=376\ \mathrm{kHz}$$

在实际的 DSP 应用中，AIC 的 MCLK 一般由 DSP 的主时钟经分频得到，例如 TMS320C30 采用 40 MHz 主时钟，经 8 分频后得到 5 MHz，若计数器 A＝8，计数器 B＝31，则

$$\mathrm{SCF} = \frac{5000\mathrm{kHz}}{8\times 2} = 312.5\,\mathrm{kHz}$$

A/D或D/A的转换频率为

$$\mathrm{SCF}=\frac{312.5}{31}=10.08\ \ \mathrm{kHz}$$

此时输入滤波器的带宽为

$$\frac{312.5}{288}\times 3.6\mathrm{kHz}=3.9\mathrm{kHz}$$

TLC32044 与 DSP 进行数据交换是通过 DX 和 DR 引脚进行的，其中 DX 引脚接收数据，DR 引脚发送数据，每次传输 16 位。在 16 位数据中，最高位在前，最低位在后。由于 A/D 和 D/A 的精度均为 14 位，因此，基本数据传输时 16 位中有效的为 14 位，位于 16 位中的高端。TLC32044 接收数据时，基本通信中的最低 2 位用于外部对 TLC32044 的控制，如果最低 2 位为 11，则在完成基本通信后启动辅助通信。在辅助通信中，外部可对 TLC32044 进行更多的控制，如发送和接收寄存器 A、B 和 A'的辅值及其他功能的控制。表 11-7 和表 11-8 分别是最后 2 位 DX 数据格式和辅助 DX 串行通信的协议。

表 11-7　DX 数据字格式的选择（最后 2 位）

d1 d0	说　明
0　0	发送和接收计数器 A 分别用 TA 和 RA 值装入，发送和接收计数器 B 分别用 TB 和 RB 值装入
0　1	发送和接收计数器 A 分别用 TA＋TA'和 RA＋RA'值装入，发送和接收计数器 B 分别用 TB 和 RB 值装入
1　0	发送和接收计数器 A 分别用 TA－TA'和 RA－RA'值装入，发送和接收计数器 B 分别用 TB 和 RB 值装入
1　1	发送和接收计数器 A 分别用 TA 和 RA 值装入，发送和接收计数器 B 分别用 TB 和 RB 值装入；在延迟 4 个移位时钟周期后，辅助的 DX 通信将立即开始，可对 AIC 编程使其工作在所要求的结构

表 11-8　辅助 DX 串行通信协议

××\|<- 至 TA 寄存器->\|××\|<- 至 RA 寄存器　-> \|0　0	d13 和 d6 是无符号二进制数的最高位
×\|<- 至 TA'寄存器 -> \|×\|<-　至 RA'寄存器　-> \|0　1	d14 和 d7 是二进制补码数的符号位
×\|<- 至 TB 寄存器　->\|×\|<-　至 RB 寄存器　-> \|1　0	d14 和 d7 是无符号二进制数的最高位
× × × × × ×　d9　× d7　d6　d5　d4　d3　d2　1　1	d2 = 0/1　去掉/接入 A/D 高通滤波器 d3 = 0/1　撤除/启动自环功能 d4 = 0/1　使 AUX IN＋和 AUX IN－引脚无效/有效 d5 = 0/1　发送和接收的异步/同步方式 d6 = 0/1　增益控制位 d7 = 0/1　增益控制位 d9 = 0/1　去掉/接入片内 2 级（sinx）/x 校正滤波器

图 11-23 是 5 V 的 C3x DSP 芯片采用缓冲串行接口与 TLC32044 连接的硬件接口示意图。从图中可以看出，TLC32044 与 DSP 之间的接口十分简单，两者之间接口不需任何硬件支撑，串行口的几根信号线直接相接，MCLK 单独提供（也可利用 DSP 的定时器产生分频时钟），WORD/$\overline{\mathrm{BYTE}}$ 引脚置高电平，选择 16 位字通信方式，C3x 芯片的 XF 作为 TLC32044 的复位信号，这样 TLC32044 的复位可由 DSP 用软件控制，从而可使 C3x 芯片的串行口在 AIC 开始进行 A/D 和 D/A 转换前完成初始化。

为使 C3x 能与 AIC 进行正确的通信，DSP 的程序必须作如下初始化工作：

（1）复位串行口并设置串行口全局控制寄存器，包括：设置串行口数据长度为 16 位、使

能串行口中断、对串行口信号进行极性控制，这里，除 BFSX 和 BFSR 设置为低有效外，其余均为高有效。

（2）设置串行口发送和接收控制寄存器。设置串行口的所有引脚工作在串行口方式，而不是通用的 I/O 方式。

（3）全局中断使能，并使串行口脱离复位状态，进入正常工作状态。

在作上述 DSP 初始化工作之前，首先置 XF0＝0，用于复位 AIC，使 AIC 设置为默认配置状态，并暂停 AIC 工作。当 DSP 的串行口初始化完成后，将 XF0 置高，使 AIC 脱离复位状态，并开始以默认配置方式工作。复位后，AIC 以默认配置方式工作，AIC 内部的各寄存器的复位状态如下：

TA＝RA＝9H
TA’＝RA’＝1H
TB＝RB＝24H

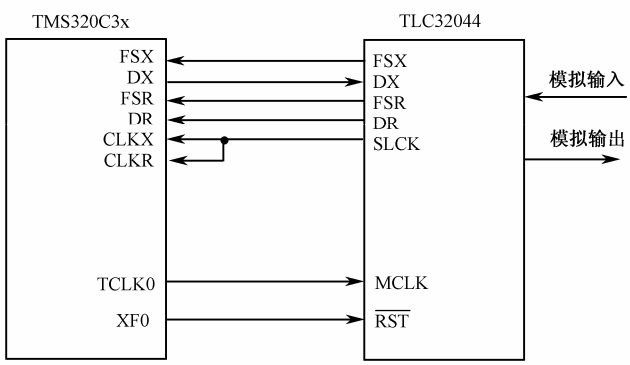


图 11-23　TLC32044 与 TMS320C3x 的硬件接口示意图

控制寄存器的各位状态为：

d9＝1，d7＝1，d6＝1，d5＝1，d4＝0，d3＝0，d2＝1

显然，AIC 在默认工作方式时采用的是同步方式。在发送和接收同步工作时，A/D 转换的定时全部从 D/A 转换定时获得，因此，发送和接收同步工作时接收计数器 A、B、寄存器 RA、RA’和 RB 可以不进行初始化，当然，对它们进行初始化不会增加程序量。

下面计算 AIC 初始化参数。AIC 工作在同步方式，A/D 采样率为 10 kHz，按 16 位字方式工作。TA＝8，TB＝31，此时开关电容滤波器的频率为

SCF＝5000/（2×8）＝312.5 kHz

A/D采样频率为

f_r＝312.5/31≈10 kHz

AIC的控制寄存器设置为0x67。根据表11-8可得初始化AIC的发送序列为：

0x0003　请求 AIC 辅助通信
0x1020　TA 和 RA 寄存器初始化
0x0003　请求 AIC 辅助通信
0x0001　TA’和 RA’寄存器初始化
0x0003　请求 AIC 辅助通信

0x3E7E　TB 和 RB 寄存器初始化
0x0003　请求 AIC 辅助通信
0x0067　控制寄存器初始化

AIC和DSP之间同步串行通信的时序如图11-24所示。

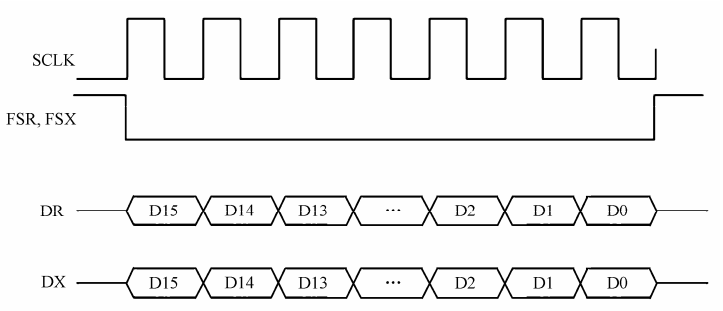


图 11-24　AIC 与 TMS320C3x 的串行通信时序

11.3.2.2　MC145483 接口的设计

MC145483 是 Motorola 公司生产的一种新型 PCM 编译码器（Codec），可以实现 13 位线性 A/D 和 D/A 变换。除了编解码功能外，芯片还有以下几个特点：

（1）具有平衡输入输出功能，增加抗噪声性能；
（2）内部设计有性能良好的电源滤波电路；
（3）配合外部增益调整，输出可直接推动 300Ω 的喇叭；
（4）采用单 3 V 供电，功耗低。

MC145483 与 C54x 芯片的接口线包括：数据输入输出线 DR 和 DT，主时钟线 MCLK，收发时钟线 BCLKR 和 BCLKT 及收发帧同步时钟线 FSR 和 FSX。Codec 与 DSP 芯片的硬件接口电路如图 11-25 所示。其中，MC145483 的主时钟和收发时钟为 2.048 MHz，帧同步时钟为 8 KHz。C54x 的串行口工作于内时钟方式，MC14LC5480 的两种时钟均由 DSP 提供。

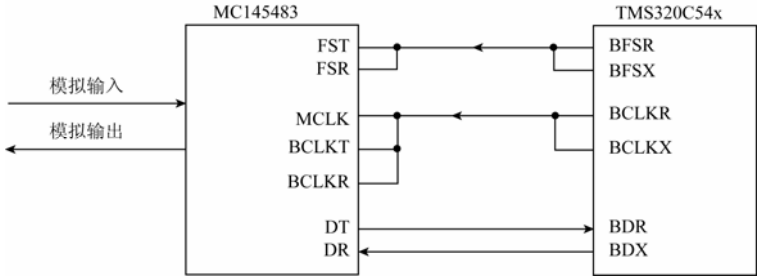


图 11-25　Codec 与 TMS320C54x 接口

11.3.3　通信接口的设计

DSP 芯片在通信系统中应用非常广泛。除了多通道缓冲串行口外，还可以利用 I/O 引脚实现外部通信。与通用的微处理器（如 80C51 系列单片机等）相比，DSP 芯片的硬件资源主要用于 DSP 处理功能，因此 I/O 引脚数相对较少。但灵活利用 DSP 芯片 I/O 资源，可以实现多种控制和通信功能。例如 TI 的 DSP 芯片一般提供有 XF 和$\overline{\text{BIO}}$两根通用 I/O 引脚，在双机通

信系统中，利用 XF 线可以中断另一个处理器，而利用$\overline{\text{BIO}}$线则可以判断外部电平的高低，从而根据对方的工作状态执行相应的子程序。本节将介绍利用 DSP 芯片 I/O 引脚实现外部通信的电路设计方法。

11.3.3.1 通信接口

在通信系统中，信息传输距离一般较远，为提高可靠性，一般不采用并行传输方式，可以利用一根数据线完成数据的按位传输。这种传输方法称为串行传输。串行传输可以分为异步传输和同步传输，一般又称为异步串行通信和同步串行通信。

异步串行通信以字符为单位进行传输。每个字符传输时，需在数据位之外增添起始位、奇偶校验位和停止位，构成一个通信帧。其帧格式如图 11-26 所示。异步串行通信按帧进行，可以连续也可以断续。

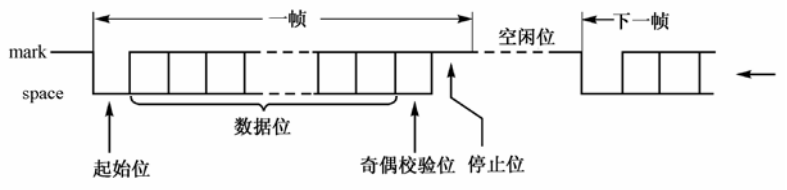


图 11-26 异步串行通信的数据格式

异步串行通信时需要不断重复起始位和停止位，占用大量通信时间，为提高传送速度，可以在数据块开始加一些特殊字符作为发送和接收双方的同步标志。同时在发送数据时给出时钟信号，以保持接收和发送的同步。这就是同步串行通信（其数据格式如图 11-27 所示）。

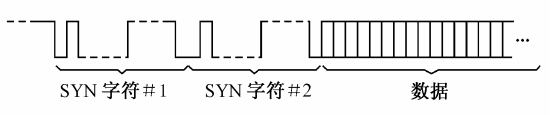


图 11-27 同步串行通信的数据格式

比较两种串行通信方式可以发现，同步通信只在数据块的开始使用同步字符串，而且数据块内各字符的格式必须相同，字符之间不允许有间隔，这个要求比异步通信更为严格，实现起来难度较大。但其传输速度快，适用于数据量大，对速度要求较高的串行通信场合。异步通信只要求每帧的时间内保持同步，实现方便，在小系统的通信中使用更多。

为适应不同场合的应用，很多公司在串行通信的基础上提出了不同的通信标准，例如串行外围设备接口总线（SPI）标准，通用串行总线（USB）标准，I^2C（Inter Integrated Circuit）总线标准，控制器局域网（Controller Area Network，CAN）总线标准等。这些标准利用简单的接口电路实现了高效的数据传输。限于篇幅，这里不对这些标准和 DSP 接口电路设计进行介绍，感兴趣的读者可以参考相关文献。

11.3.3.2 同步通信接口电路设计

在通信终端设备中，同步串行通信是常用的一种通信方式。例如 MODEM 中采用的 V.24 通信标准就是一种典型的同步串行通信方式，在这种通信方式中，主要的通信信号包括串行收发及其同步时钟，利用 TMS320C54x DSP 芯片附加一个非门就可以实现与外部（如 MODEM）

设备的同步串行通信。图 11-28 是 DSP 芯片实现的同步接口示意图。

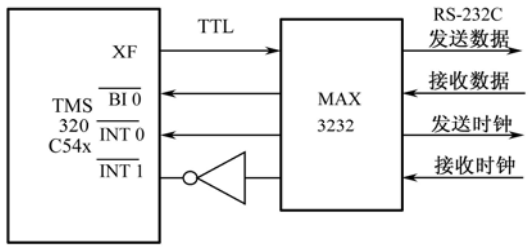


图 11-28　同步串行接口电路

在 V.24 接口标准中，串行数据（TXDATA）的发送是与外部提供的发送时钟（TXCLK）同步的，每一个比特与发送时钟的下降沿对齐，也就是说，每一个比特起始于时钟的一个下降沿而终止于下一个时钟的下降沿。因此，为保证可靠接收，应该在时钟的上升沿对接收数据进行采样。

图 11-28 中，XF 引脚和$\overline{\text{BIO}}$引脚分别作为串行发送和接收的数据线，中断引脚$\overline{\text{INT0}}$与外部提供的串行发送时钟（TXCLK）相接，而外部提供的串行接收时钟（RXCLK）经过一个非门与$\overline{\text{INT1}}$中断引脚相接。数据发送是由 INT0 中断程序完成，当 TXCLK 下降沿到来时，触发一次 INT0 中断，DSP 响应 INT0 中断并在中断程序内发送一个比特。与此类似，当 RXCLK 的上升沿到来时，经过反相形成下降沿触发 DSP 的 INT1 中断，DSP 响应中断并在中断程序内采样$\overline{\text{BIO}}$引脚的状态从而完成一次比特接收。

图 11-28 中同步串行数据和时钟利用 RS-232C 标准接口传输，芯片 MAX3232 完成 RS-232C 到 TTL 的电平转换功能。

11.3.3.3　异步通信接口电路设计

在需要与外部异步通信器件接口时，利用 XF 和$\overline{\text{BIO}}$也可以实现一个异步串行口（需要内部定时器和一个中断引脚）。用 XF 作为发送数据线，而用$\overline{\text{BIO}}$作为接收数据线，异步通信格式则完全用软件实现。这种方法在通信速率较低时或仅在开机阶段用作程序引导时比较合适。如果异步通信速率高且需要灵活控制，则一般需外加接口芯片（如 TL16C550C 等）来实现。图 11-29 是用 XF 和$\overline{\text{BIO}}$线实现异步通信接口的示意图。

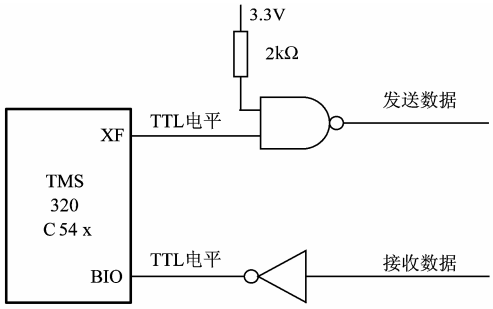


图 11-29　异步通信接口的示意图

11.3.4　主从式系统的设计

在 DSP 应用系统中，有些应用系统采用单个 DSP 芯片辅以必要的存储器、模数转换器和外

设接口等就可以实现。但在有些应用场合，由于系统除了要完成数字信号处理功能外，还有许多诸如与外部系统的通信、控制、人机接口等功能，如果仅用一个 DSP 来实现，或者是由于处理能力所限不能完成包括数字信号处理在内的所有功能，或是使硬件系统过于复杂。在这种情况下，往往需要采用多个 DSP 或是一个 DSP 加一个通用的微处理器（MPU）来实现，每个处理器完成系统的一部分工作。一般来说，如果系统的数字信号处理任务由一个 DSP 芯片能够实现，那么，用一个 MPU 加上一个 DSP 构成主从式系统是一个比较理想的方案，DSP 芯片主要完成系统的数字信号处理功能，而 MPU 则完成系统的其他功能，包括与外部的通信、控制等功能。在这种主从式硬件系统中，一般 MPU 作为主机，而 DSP 则作为从机，主机可以控制从机的复位、运行和挂起，从机在主机的控制下完成所分担的一部分工作。主机将从外部获得的数据交由从机处理，而从机则将处理后得到的有关结果传递给主机，由主机将处理结果发送至其他系统。

主从式硬件系统设计的关键是主机与从机之间的数据通信。一般实现双机通信的方式主要有串行通信、并行通信和共享存储器几种方式。其中串行通信相对来说比较简单，适合于双机通信量不是很大的应用场合。并行通信利用处理器的 I/O 功能，在 MPU 和 DSP 之间增加缓冲器或锁存器实现双机通信，与串行通信方式相比，效率稍高些。共享存储器实现时可以分为两种，一种是主机和从机不能同时访问共享存储器，另一种是主机和从机可以同时访问共享存储器（当然不能同时访问同一个单元）。前一种共享存储器方式通常是利用 DSP 提供的 DMA 功能，而后者则常采用双端口存储器来实现。下面我们来讨论这些通信方式。

11.3.4.1 并行通信实现双机通信

在主机和从机之间通信量不是太大的系统中，采用双向锁存并行通信是实现双机通信的一种比较合适的方式。下面以 8 位双向锁存为例讨论双机通信的实现方法。

图 11-30 是采用 8031 单片机作为主机，C50 芯片作为从机实现 8 位双向锁存的硬件示意图。图中，8 位双向锁存用 2 片 74F574 完成，其中 74F574（1）用于锁存 8031 至 C50 芯片的 8 位数据，74F574（2）用于锁存 C50 芯片至 8031 的 8 位数据。同时 8031 的 P1.0 引脚接至 C50 芯片的$\overline{\text{INT1}}$引脚，用于 8031 向 DSP 申请 INT1 中断，而 C50 芯片申请 8031 INT1 中断是通过 XF 引脚实现的。

双机通信的过程如下：当 8031 有数据需要向 C50 芯片发送时，首先将数据锁存在 74F574（1）中，然后在 P1.0 引脚产生一个低电平脉冲申请中断，C50 芯片响应中断，并在 INT1 中断服务程序中将锁存在 74F574（1）中的数据读入，从而完成了 8031 向 DSP 的数据传送。同样，当 C50 芯片需将数据发送至 8031 时，首先将 8 位数据锁存在 74F574（2）中，然后在 XF 引脚上置一低电平脉冲申请 8031 中断，8031 响应中断并在中断服务程序中读出锁存在 74F574（2）中的数据，从而完成 DSP 向 8031 的数据传送。

图 11-30 中$\overline{\text{RD1}}$和$\overline{\text{WR1}}$ 分别是 8031 读、写锁存器的选通信号，而$\overline{\text{RD2}}$和$\overline{\text{WR2}}$分别是 C50 芯片的读、写锁存器的选通信号，它们映射至 DSP 的 I/O 空间。由于锁存器的 8 位数据接至 C50 芯片的低 8 位数据线，因此 C50 读或写锁存器时采用一个 16 位字的低 8 位，高 8 位数据无效。如果 MPU 的 I/O 数据是 16 位，则增加两片 74F574 就可以实现 16 位的双向通信。

在需要节省体积的应用系统中，也可以用一片 74F652 代替两片 74F574。

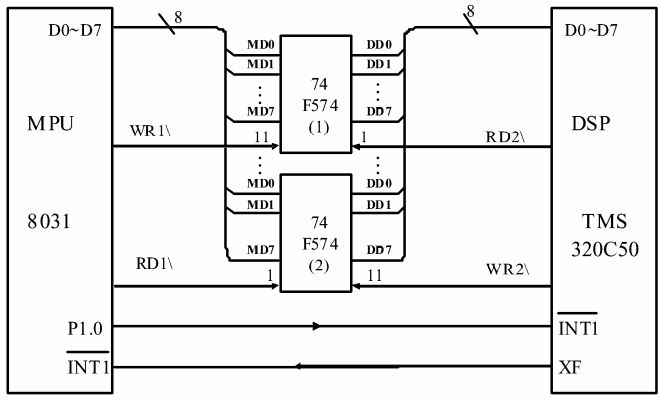


图 11-30　8031 与 TMS320C50 的并行通信

11.3.4.2　共享存储器实现双机通信

共享存储器可以采用 DMA 方法实现双机通信的优点是适合通信量大的应用场合，但缺点是双机不能同时访问存储器，在某一时刻存储器只能作为某一个处理器的存储器。而用并行锁存方法实现的双机通信可以做到真正的并行通信，但通信量受到限制。解决上述两种方法缺点的一种双机通信方法是采用双端口 RAM。

与普通的 RAM 不同，双端口 RAM 是一种特殊的存储器，它具有两组数据总线和地址总线，两组总线可以同时访问不同的存储器单元，当两组地址总线完全相同时，由片内总线仲裁逻辑向后访问的一方发出等待信号，使该方进入等待，待另一方访问结束之后等待撤销，等待方继续访问这一地址。由于双端口 RAM 的特殊结构，使得双机可以方便快速地进行数据交换，从而大大提高了 MPU 与 DSP 芯片的并行处理能力。

下面介绍一种双端口 RAM 芯片 IDT71342 及其工作原理。这种芯片由 IDT 公司推出，容量 4K×8 比特，利用它可以不加任何逻辑构成 4K×8 位的双端口存储器系统。如果要构成 4K×16 位，采用两片 IDT71342 即可。

IDT71342 具有两组独立的数据总线、地址总线和相应的控制信号线，它允许主、从两个 CPU 同时访问双端口 RAM 中的不同单元。双端口 RAM 的两个端口与普通的 RAM 一样，可读可写。唯一可能发生冲突的是两个 CPU 对双端口 RAM 的同一个存储单元同时读或同时写。为了避免这种情况的发生，双端口存储器采用所谓的 Semaphore 信号（以下简称 SEM 信号）来避免这种冲突。SEM 逻辑是与双端口 RAM 存储单元无关的 8 个单元，这些单元用来产生并存储一个标志，以向另外一侧说明共享的资源一侧正占用。IDT71342 的 SEM 逻辑如图 11-31 所示。

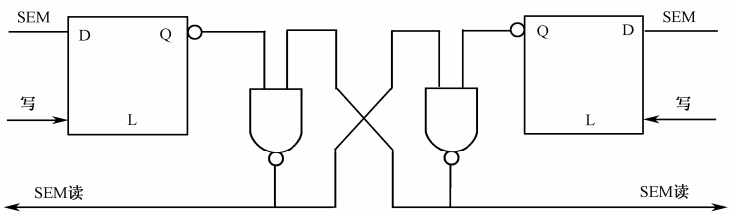


图 11-31　SEM 信号逻辑

SEM 信号为低电平有效，将“0”写入寄存器申请控制权，写入“1”释放控制权。如果对没有使用过的 SEM 寄存器一侧写入“0”，则写入侧读出也为“0”，而另一侧读出则为“1”。此时 SEM 信号只有读入为“0”的一侧才能修改，如果该侧写入“1”，则两侧寄存器都置为“1”，之后两侧都可以设置 SEM 信号。如果一侧已经设置了 SEM 信号，另一侧再设置，则另一侧的申请一直锁存直至获得控制权的一侧放弃申请，另一侧才能获得控制权。表 11-9 是 SEM 信号的应用例子。

将 4K×8 位分成多少块以及块的大小可以完全由软件决定。最简单的例子是将 4K×8 位分为两个 2K×8 为的 RAM 区。这样双方都可以获得一定资源的 RAM 区。采用两个 SEM 寄存器用来指示两块 RAM 被哪一侧的 CPU 所占用。可使用 SEM_A 指示低 2K×8 RAM 被哪一侧 CPU 占用，而使用 SEM_B 指示高 2K×8 RAM 被哪一侧 CPU 占用。设一侧为 MPU，另一侧为 C54x DSP。若 MPU 向 SEM_A 写入“0”，然后读出也是“0”，则说明 MPU 可以读写低 2K×8 RAM，低 2K×8 被 MPU 控制。如果读出为“1”，则说明低 2K×8 被 C54x 芯片所占用。同样，也可以对 SEM_B 进行相同的处理。MPU 完成低 2K×8 RAM 操作后，对 SEM_A 写入“1”，以释放 2K×8 的控制权。由此可见，SEM 逻辑能增加双机系统的数据交换能力。采用双端口 RAM 是实现双机通信的一种比较理想的方法。

表 11-9　SEM 信号的应用

功　能	SEML	SEMR	状　态
无操作	1	1	共享区空闲
左端写“0”	0	1	左端获控制权
右端写“0”	0	1	不变，右端未获控制权
左端写“1”	1	0	右端获控制权
左端写“0”	1	0	不变，左端未获控制权
右端写“1”	0	1	左端获控制权
左端写“1”	1	1	共享区空闲
右端写“0”	1	0	右端获控制权
右端写“1”	1	1	共享区空闲
左端写“0”	0	1	左端获控制权
左端写“1”	1	1	共享区空闲

11.3.4.3　利用 HPI 实现双机通信

HPI 模块可以用于和外部主机实现并行通信，关于模块的配置和使用已在第 9 章进行介绍，这里介绍一种利用 HPI-8 实现 TMS320C54x 与单片机实现通信的接口方法。

图 11-32 是单片机和 DSP 的 HPI 口硬件接口电路图。单片机为双机系统的主控部分，采用端口方式直接访问 HPI 口。双方通过共享 TMS320C54x 的片上存储器完成通信。

对 HPI 口进行数据读写操作需要三个步骤：设置控制寄存器，写地址寄存器，读写数据寄存器。在寄存器读写过程中，单片机通过端口发送控制信号，检测状态信号，完成对 HPI 口访问的时序模拟。

HPI 口的存储器访问有共用寻址（SAM）和单主机寻址（HOM）两种工作方式。在 SAM 方式下，主机和 DSP 都能访问 HPI 存储器；在 HOM 方式下，只有主机才能访问 HPI 存储器。一般采用共用寻址方式。整个通信过程以各部分状态的变化来推进。DSP 和单片机通过向对方发送中断通知对方数据已准备好，通过检测对方设置的状态判断对方是否准备好接收数据。

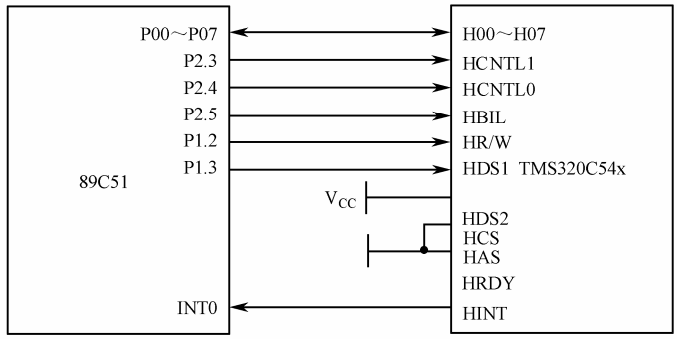


图 11-32　单片机和 DSP 的 HPI 口硬件接口

以下是单片机和 DSP 的通信程序。

1．DSP的通信程序

```
                .mmregs
                .def    main, HPI_ISR
                .data
Tx_data:    .word 0,1,2,3,4,5,6,7    ;供单片机读取，地址为 80h~87h
Rx_data:    .usect "rdata",8         ;供单片机写入，地址为 88h~8fh
                text
main:
                ;set stack
                STM             #90h,SP     ;堆栈地址 90h~94h
                ADDM            #4h,SP
                ANDM            #0fffeh,SP
                ;set CLKMD              ;10 倍频
                STM             0,CLKMD
W_CLK:          LDM             CLKMD,A
                AND             #1,A
                BC              W_CLK,ANEQ
                STM             #9207h,CLKMD
                ;relocate the interrupt table  ;开 HPI 中断
                ANDM            #7fh,PMST
                ORM             #1180h,PMST
                ;set HPIC
                ;set IFR,IMR
                STM             0,IFR
                ORM             200h,IMR
                RSBX            INTM
                ;set ptr
                STM             #Rx_data,AR2
                ;send    HPI INT to C51
                STM             0ah,HPIC
loop:           B               loop
```

```
HPI_ISR:
                ;count the data sent by c51
                RPTZ        A,7
                ADD         *AR2+,A
                POPM        ST1
                POPM        ST0
                RETE
            .end
```

2．单片机通信程序

头文件 HPITEST.H：

```
#include <REG51.h>
sbit HRW        =       P1^2;
sbit HCNTL1     =       P2^3;
sbit HDS1       =       P1^3;
sbit HCNTL0     =       P2^4;
sbit HBIL       =       P2^5;
sbit HOE        =       P2^6;
sbit HRESET     =       P1^4;
#define INT0_VECT 0
```

主程序：

```
#include <HPITEST.H>
void C51_Init（void）;        // c51 初始化函数
void HPI_Init（void）;        // HPI 初始化函数
void HPIC_Write（void）;   // HPIC 主机配置函数
void HPID_Write（void）;   //  主机向 DSP 写数据函数
void HPID_Read（void）;   //  主机从 DSP 读数据函数
void HPIA_Write（void）;   //  主机设置内存地址函数
unsigned char   GetData[2];  //2 byte data
bit Flag_Hint;                  //interrupt flag
void main （void）
{
            C51_Init（）;
            HPI_Init（）;
            EA = 1;                  //使能所有中断
            while （1）              //主循环
            {
                if（Flag_Hint == 1）
                {
                    unsigned int i;
                    int l_data[16];
                    GetData[0] = 0x80;              //DSP 存储数据地址
                    GetData[1] = 0x00;
                    HPIA_Write（）;
                    for（i=0;i<0x8;i++）
```

```
                {
                    HPID_Read（）;
                    l_data[2*i+1] = GetData[1]; //保存接收数据
                    l_data[2*i] = GetData[0];
                    }
                GetData[0] = 0x87;      //单片机写入数据地址。以地址连续方式写入时，
                GetData[1] = 0x00;      //地址先增 1
                HPIA_Write（）;
                for（i=0;i<0x8;i++）
                {
                  GetData[1] = l_data[2*i+1];
                  GetData[0] = l_data[2*i];
                  HPID_Write（）;
                }

                  GetData[0] = 0x0c;          //清除 HPI 中断，触发 DSPINT  中断
                  GetData[1] = 0x0c;
                  HPIC_Write（）;
                  Flag_Hint = 0;              //复位标志
      }
     }
}

void C51_Init（void）
{
    //enable int0 and configure the interrupt type
     IT0 = 0;                 //low level trigger
     EX0 = 1;                 //enable the int0 interrupt
     Flag_Hint = 0;           //initialize the flags
}

void HPI_Init（void）
{
     HOE = 0;          //to enable the boot
    //to do:intialize the HPIC
     GetData[1] = 0x08;
     GetData[0] = 0x08;
     HPIC_Write（）;       //clear the intial HPIC interrupt flag and set BOB
}
void int0_isr（void） interrupt INT0_VECT
{
//judge if it is a valid interrupt
     GetData[1] = 0x00;
     GetData[0] = 0x81;
     HPIA_Write（）;
     HPID_Read（）;
```

```
            if（GetData[0]==0x01 && GetData[1]==0x00）
            Flag_Hint = 1;     //set the interrupt flag
            GetData[1] = 0x08;
            GetData[0] = 0x08;
            HPIC_Write（）;        //clear the intial HPIC interrupt flag
            }
void HPID_Read（void）
{
      HRW = 1;
      HDS1 = 1;
      HRW = 1;
      HBIL = 0;
      HCNTL1 = 0;                  //连续地址访问方式
      HCNTL0 = 1;
      HDS1 = 0;
      ACC = 0;                //delay a clock;
      GetData[1] = P0;
      HDS1 = 1;
      HRW = 1;
      HBIL = 1;
      HCNTL1 = 0;
      HCNTL0 = 1;
      HDS1 = 0;
      ACC = 0;                //delay a clock;
      GetData[0] = P0;        //read the lower byte
      HDS1 = 1;
}
void HPIC_Write（void）
{
      HRW = 0;
      HDS1 = 1;
      HRW = 0;
      HBIL = 0;
      HCNTL0 = 0;
      HCNTL1 = 0;
      P0 = GetData[1];
      HDS1 = 0;
      ACC = 0;                //delay a clock
      HDS1 = 1;
      HRW = 0;
      HBIL = 1;
      HCNTL0 = 0;
      HCNTL1 = 0;
      P0 = GetData[0];
      HDS1 = 0;
```

```
        ACC = 0;                //delay a clock
        HDS1 = 1;
}
void HPID_Write（void）
{
        HRW = 0;
        HDS1 = 1;
        HRW = 0;
        HBIL = 0;
        HCNTL1 = 0;
        HCNTL0 = 1;
        P0 = GetData[1];
        HDS1 = 0;
        ACC = 0;                //delay a clock
        HDS1 = 1;

        HRW = 0;
        HBIL = 1;
        HCNTL1 = 0;
        HCNTL0 = 1;
        P0 = GetData[0];
        HDS1 = 0;
        ACC = 0;                //delay a clock
        HDS1 = 1;
}
void HPIA_Write（void）
{
        HRW = 0;
        HDS1 = 1;
        HBIL = 0;
        HCNTL1 = 1;
        HCNTL0 = 0;
        P0 = GetData[1];
        HDS1 = 0;
        ACC = 0;                //delay a clock
        HDS1 = 1;

        HRW = 0;
        HBIL = 1;
        HCNTL1 = 1;
        HCNTL0 = 0;
        P0 = GetData[0];
        HDS1 = 0;
        ACC = 0;                //delay a clock
        HDS1 = 1;
}
```

程序说明：

（1）DSP 通信程序功能。DSP 通信程序将要发给单片机的数据放于地址为 80h~87h 的 8 个存储单元。在设置好 DSP 的状态后，DSP 向单片机发中断，通知单片机，DSP 已将数据准备好。并等待单片机发中断，表示单片机已将数据送入。DSP 在中断中对接收的数据进行处理。

（2）单片机通信程序功能。单片机通信程序在完成初始化之后，等待 DSP 发中断通知单片机数据已准备好。单片机在检测到中断后，先判断中断是否有效，再从 HPI 口读写数据。在完成向 HPI 口发送数据后，向 DSP 发中断通知 DSP 接收数据。单片机写入数据的地址为 88h~8fh。

（3）单片机读写 HPI 口寄存器说明。单片机通过控制端口信号模拟接口时序，来完成对 HPI 口寄存器的访问。具体控制时序可以参考程序中访问寄存器子程序的操作。

（4）初始化说明。因为 DSP 在 Boot 的过程中向单片机发送了 HPI 中断。单片机在初始化时要清除这个中断。并且，主机在数据交互之前要将控制寄存器中的 BOB 位设置好，指示高地址在前还是低地址在前。BOB=1 表示第一个字节是高地址字节。这一步也在程序初始化时由单片机完成。因为 DSP 复位后默认处于共享模式（SAM）,所以 DSP 通信程序中没有对存储器访问模式进行设置。

11.4 小　　结

本章讨论了 DSP 硬件系统设计的常用方法，包括复位电路、电源电路以及硬件设计的可靠性技术等，在此基础上介绍了外部存储器接口电路、模数接口电路、通信电路和主从式系统的一般设计方法。掌握了这些基本硬件设计方法和典型电路的设计，读者就可以根据实际应用设计一个独立完整的的 DSP 硬件系统。

11.5 习题与思考题

（1）一个 DSP 最小硬件系统一般包括哪些部分？

（2）一个 DSP 系统中采用 TMS320VC5402 DSP 芯片，系统中还有若干 5 V 器件，试为该系统设计合理的电源。

（3）如何保证硬件系统的稳定可靠？主要有哪些方法？

（4）如何设计 DSP 芯片的外部存储器接口？TMS320C54x 和 TMS320C55x 芯片的存储器接口设计有何区别？

（5）如何设计 DSP 芯片的模数接口电路？如果模数转换芯片不是串行接口，想一想如何设计与 DSP 芯片之间的接口？

（6）通用异步收发（UART）是电子系统中经常使用的异步串行通信方式，由于其接线简单，协议实现方便得到了广泛的应用。请查阅一些参考资料，利用通用 I/O 接口，完整地设计 UART 电路。

（7）实现双机通信有哪些方法？各种方法的基本原理是什么？

（8）利用 TMS320C54x 的 HPI 接口，设计实现 89C51 单片机与 DSP 芯片之间的通信。

第 12 章　DSP 芯片应用系统开发实例

12.1　引　　言

前面几章介绍了 DSP 芯片的开发工具、基于 C 语言和汇编语言的 DSP 开发以及 DSP 系统的软硬件设计。本章我们以 TMS320C203、TMS320C31 和 TMS320VC5409 为例介绍定点 DSP 芯片和浮点 DSP 芯片应用系统的设计、调试和开发过程。

12.2　一个基于 TMS320C203 DSP 应用系统的开发

TMS320C203 是 TMS320C2000 系列中一种低成本的 DSP 芯片，其最快的运算速度是 40MIPS，其源代码与早期的 TMS320C2x 兼容，代码与 TMS320C5x 兼容。本节介绍 TMS320C203 应用于语音处理的一个应用系统——数字压缩语音录放系统，着重介绍这个系统的软硬件设计、调试和独立系统形成等开发过程。

12.2.1　系统简介

本系统实现的是一个基于定点 DSP 芯片 TMS320C203 的数字压缩语音录放系统，其主要功能是通过对语音信号进行压缩以实现高效率数字录音，可应用于电话留言、语声应答等场合，其中电话留言是本系统的一个典型应用。采用磁带录音实现电话留言，虽然录音的时间较长，但不便于查找和保存。数字录音可以克服磁带录音所存在的缺点，不仅查找速度快，而且对录音信息进行编辑整理也非常方便，更为方便的是对数字录音的信息可以转存在计算机硬盘或光盘上以便长期保存。但数字录音也存在一个突出的缺点，即需要较大的存储空间。例如，若采用 64 kbit/s 的 PCM 编码方式，则存储 1 min 的语音需要 480 KB 的存储空间，存储 10 min 的语音则需要的存储空间为 4.8 MB。显然，要实现较长时间的录音需要的存储空间将更大。正因为如此，市场上推出的具有数字留言功能的电话机的录音时间一般都较短。

实现长时间数字录音的方法有两种，一是扩大存储空间，但无限地扩大存储空间势必增加系统成本；二是采用语音压缩的方法，即首先将输入的语音进行压缩，然后存储在存储器中。本系统结合这两种方法，一方面采用存储量为 16 MB 的单片闪速存储器 KM29N160，另一方面采用 2.0 kbit/s 的速率对语音进行压缩。采用这两种方法后，可以实现 2 h 以上的数字录音。

考虑到系统的成本和功耗等因素，系统中实现语音压缩算法由性能价格比较高的定点 DSP 芯片 TMS320C203 实现，AT89C51 单片机作为主控器完成系统的控制和键盘处理、显示等功能。闪速存储器与 TMS320C203 相接，便于 TMS320C203 存入和读出压缩数据。整个系统具有录音、放音、快进、快倒、暂停、停止、清除、数字音量调节等功能，使用比较方便。下面介绍本系统的软硬件设计及调试方法。

12.2.2 系统硬件设计

1．设计考虑

整个硬件系统包括三个部分。一是 TMS320C203 DSP 处理系统，包括 TMS320C203、程序存储器、数据存储器、模数转换电路等；二是闪速存储器及其与 TMS320C203 接口部分；三是 AT89C51 主处理器部分，包括 AT89C51 处理器、键盘显示电路及 AT89C51 与 TMS320C203 接口。为了降低系统成本，在 DSP 处理系统中，采用 EPROM 和高速静态 RAM 相结合的存储器配置方式。其中，EPROM 用于存储程序和初始化数据，高速 RAM 用于程序执行和数据的暂存，程序和数据共用两片高速 RAM，简化了系统硬件。图 12-1 是整个硬件系统的示意框图。下面分别简要介绍这三个部分的硬件设计。

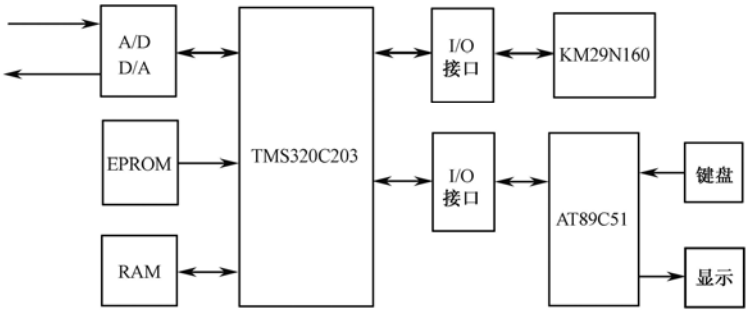


图 12-1　压缩录放系统硬件示意图

2．DSP处理系统

TMS320C203 DSP 处理系统主要完成语音的压缩和解压缩功能。系统主要由 TMS320C203、EPROM 27C256-15（1 片）、高速 RAM CY7C199-10（2 片）、译码电路、晶体振荡器、模数转换电路等构成。其中，EPROM 存储 Boot 信息和程序代码，其地址为数据空间的 8000H~FFFFH，DSP 芯片加电运行时将 EPROM 中的程序代码搬移到高速 RAM 中 。高速 RAM 的存取时间为 10 ns，可以全速执行，地址为 0000H~7FFFH，共 32 K 字，程序和数据均可访问。

模数转换采用 Motorola 公司的 PCM 编解码器 MC14LC5480，这个芯片集语音 A/D、D/A 及抗混叠滤波于一体，采用单 5 V 供电方式，使用比较方便，性能价格比较高。由于 TMS320C203 提供了可与 PCM 编解码器等串行器件接口的串行口，因而两者接口十分方便。TMS320C203 的串行口包括数据输入 DR、数据输出 DX、接收时钟 CLKR、发送时钟 CLKX、接收同步时钟 FSR 和发送同步时钟 FSX。MC14LC5480 与 TMS320C203 接口时，只需将相应的引脚一一连接即可。

3．闪速存储器接口电路

图 12-2 是闪速存储器 KM29N160 及其与 TMS320C203 的接口电路示意图。KM29N160 具有一个可复用的 8 位 I/O 口，数据和地址的输入输出及命令的输入都是通过这个 8 位 I/O 口实现的。此外，还有若干控制信号线用于控制芯片的各种操作和指示芯片工作状态。这些信号线包括：命令锁存使能线 CLE，地址锁存使能线 ALE，片选线 $\overline{\text{CE}}$，读使能线 $\overline{\text{RE}}$，写使能线 $\overline{\text{WE}}$，

R/$\overline{B}$ 线指示芯片的工作状态，WP 线提供上电期间意外写保护功能。

数据的读写和芯片的擦除等功能是通过一系列的串行操作完成的。例如，从芯片中读 32 字节的一系列操作包括：写入读命令字，写入 A0~A7 地址，写入 A8~A15 地址，写入 A16~A23 地址，读 32 字节。其他功能的操作与此类似。

与 TMS320C203 的接口由 1 片 74F138 译码器、1 片 74HC244 缓冲器和 2 片 74HC374 锁存器构成。其中，74F138 实现 I/O 地址译码功能，产生 I/O 读写信号，$\overline{\text{FMI/O_RD}}$ 是从 74HC244 读取数据的读使能线，$\overline{\text{FMI/O_WE}}$ 是将数据写入锁存器的写使能线，$\overline{\text{FMC_WE}}$ 是将控制命令写入锁存器的写使能线。74HC244 用于缓冲从闪速存储器至 TMS320C203 的 8 位数据，74HC374（1）用于锁存 TMS320C203 至闪速存储器的 8 位数据，74HC374（2）则用于锁存 TMS320C203 输出至闪速存储器的各种控制信号。TMS320C203 的 $\overline{\text{BIO}}$ 线通过上拉电阻接至闪速存储器的 R/$\overline{B}$ 线，用于读取闪速存储器的工作状态。

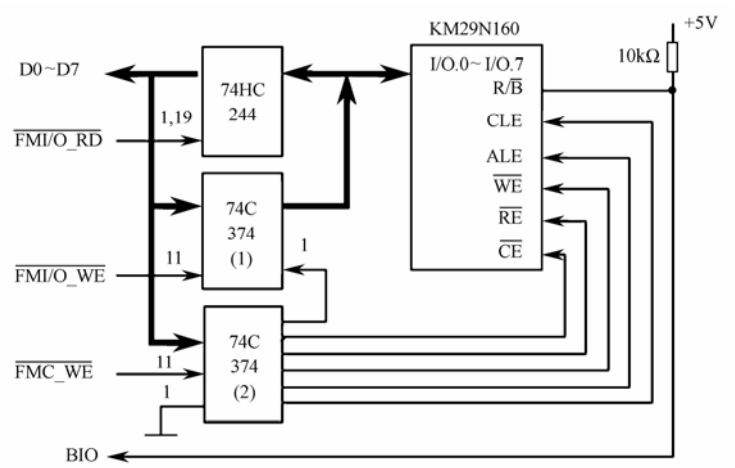


图 12-2　闪速存储器接口

4．主从系统电路设计

整个硬件系统由 AT89C51 和 TMS320C203 两个处理器以主从方式连接实现。其中 AT89C51 作为系统主机，完成系统的人机接口（键盘、显示）和对 TMS320C203 的控制。89C51 的 P0 口用作 8 个按键输入，串行口用做数码管的显示接口，P2.0 接至 TMS320C203 的 $\overline{\text{INT0}}$ 中断，P2.1 用于复位 TMS320C203，P2.2 是接口芯片 74HC374 的输出使能线，P2.3 和 P2.4 为两个发光二极管的控制线，P2.5 为速率选择线。AT89C51 的 P1 口用于与 TMS320C203 之间的双向数据交换。其中，TMS320C203 向 AT89C51 发送数据通过 74HC374 锁存器实现，而 AT89C51 向 TMS320C203 发送数据则通过 74HC244 缓冲器实现。AT89C51 的 $\overline{\text{INT0}}$ 中断线接至 TMS320C203 的 XF 引脚。图 12-3 是 AT89C51 主处理器与 TMS320C203 之间的接口电路示意图。其中，$\overline{\text{MPU_WE}}$ 是 TMS320C203 将数据写入 74HC374 锁存器的使能线，$\overline{\text{MPU_RD}}$ 是 TMS320C203 从 74HC244 读取数据的使能线，这两根线均由 74F138 译码而得，并且映射在 TMS320C203 的 I/O 空间。

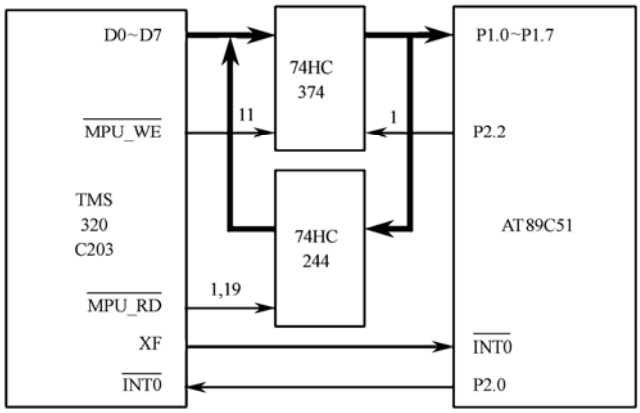


图 12-3　AT89C51 与 TMS320C203 的接口

AT89C51 与 TMS320C203 采用 8 位方式与 TMS320C203 交换数据。AT89C51 向 TMS320C203 发送数据时，首先将数据输出至 P1 口，然后在 P2.0 引脚上产生一个低电平脉冲向 TMS320C203 发 INT0 中断，TMS320C203 响应 INT0 中断并在中断服务程序中读取 8 位数据。TMS320C203 发送数据至 AT89C51 时，先将 8 位数据锁存至接口芯片 74HC374 中，然后在 XF 线上产生一个低电平脉冲向 AT89C51 发 INT0 中断，AT89C51 在其中断服务程序中通过 P2.2 线读取锁存在 74HC374 中的 8 位数据。

12.2.3　系统软件设计

1．TMS320C203 软件

TMS320C203 的软件包括 EPROM 引导程序和系统软件两部分。其中，EPROM 引导程序比较简单，其作用是将系统软件从低速 EPROM 搬移至高速 RAM 中。系统软件由三部分组成，即主程序、串行中断服务程序和 INT0 中断服务程序。主程序完成系统的初始化，包括从闪速存储器中读取已经存入的语音段数及数字音量等信息。完成初始化过程后，TMS320C203 就等待从 AT89C51 主处理器发来的各种命令，根据不同命令调用相应的处理程序。

串行中断服务程序完成语音的输入输出和数据格式的转化等功能，由于 PCM 编解码器输出的数据是 8 位 μ 律数据，因此需将 μ 律数据转化为线性格式的数据以便 TMS320C203 进行处理。同样，TMS320C203 输出的线性数据也需转化为 8 位 μ 律才能送至 PCM 编解码器进行 D/A 变换。

INT0 中断服务程序主要用于接收从 AT89C51 送来的各种命令，并且设置相应的命令标志以便 TMS320C203 在主程序中识别并调用相应的子程序。

2．AT89C51 软件

AT89C51 软件主要由初始化程序、主程序、定时中断服务程序和 INT0 中断服务程序等组成。在初始化程序中，完成 AT89C51 的初始化、TMS320C203 的复位、从 TMS320C203 获得已存入闪速存储器的语音段数和数字音量等，此外还将当前选择的语音压缩速率送至 TMS320C203。主程序主要是按键处理程序，完成各种按键的分析处理，并且将有关信息送至 TMS320C203 以实现相应的功能。定时器中断服务程序每隔 10 ms 中断一次，增加程序所需的各种计数器的值，并置各种定时到达标志以便主程序判断使用。此外，还有显示程序，包括数码管和发光二极管等的显示。



12.2.4　硬件调试

实现一个实时 DSP 硬件系统一般包括电路设计、印制电路板设计和硬件调试三个阶段。在电路设计阶段，关键是要保证硬件电路能够实现所要达到的 DSP 功能，在印制电路板设计阶段，关键是要使布线正确合理。布线正确一般都能做到，但要做到合理，则并不容易。DSP 硬件系统中最易出现的问题是高频干扰，因此，在布线时应尽量使高频线短而粗，且远离易受干扰的信号线，如模拟信号线。此外，电源滤波、模拟线与数字线分离等也是不容忽视的。

设计并加工好印制电路板后就可以进入硬件调试阶段。在这个阶段，首先应对电路板作细致的常规检查，防止短路和断路现象。加电后，先检查晶体是否振荡，复位是否正确可靠，用示波器或逻辑笔检查 TMS320C203 的 CLKOUT 信号是否正常，若正常表明 TMS320C203 工作基本正常。在做完这些基本检查之后，就可以进行系统硬件调试。

硬件调试一般需要用 TMS320C203 仿真器对硬件系统进行实时仿真调试。一般可按照从易到难的原则分步骤进行调试。

1．调试DSP+EPROM

首先调试最基本的 DSP 与 EPROM 的接口。调试 DSP 加上 EPROM 的基本方法就是编写一个简单的调试程序，将这个程序写入 EPROM（加上 BOOT 信息使程序 BOOT 到 DSP 的内部 RAM 中）并将 EPROM 插上电路板，加电运行后用示波器或简单的逻辑笔来观察 TMS320C203 的有关信号线以判断系统工作是否正常。TMS320C203 提供的外部标志 XF 是一个最易观察的信号线。调试程序可将 XF 置高置低使 TMS320C203 在 XF 线上输出一个高低电平变化的脉冲信号。如果不能得到预期的脉冲信号，可以使用仿真器单步执行观察 BOOT 过程以确定故障原因。下面给出的例子是在 XF 引脚上输出一个方波信号的简单调试程序。

【例 12.1】在内部 RAM 中运行，在 XF 引脚上输出方波信号

```
                .global   _main
;定义:          page 0
TST0            .set      60h
TST1            .set      61h
TMPD            .set      62h
TMPA            .set      63h

;矢量表
                .sect   ".vect"          ; 自定义块
RS              B         _main
INT1            B         _main
INT2_3          B         _main
TINT            B         _main
RINT            B         RINT_START
XINT            B         _main
TXRXINT         B         _main

;主程序
_main:          DINT                     ; 关中断
```

```
        SSXM            ; 置符号扩展方式
        CNFD            ; B0 块为数据区
LOOP:   RPTK    255
        SXF             ; XF 置高
        RPTK    255
        RXF             ; XF 置低
        B       LOOP    ; 循环
```

2．调试DSP+EPROM+RAM

在调试通过 DSP 加上 EPROM 的电路之后，可以进一步调试高速 RAM 与 DSP 的接口电路。调试高速 RAM 的方法是将例 12.1 的调试程序 BOOT 到高速 RAM 中运行，只需改变 Boot 的目的地址即可。如果 XF 输出正常，则表明 RAM 与 DSP 的接口电路也正常。

3．调试DSP+EPROM+A/D、D/A

上面两步调试完成后，DSP 系统硬件的主体运行基本正常。在上述程序的基础上，进一步调试模数转换器与 DSP 的接口。在调试这个接口之前，首先要保证 PCM 编译码器本身硬件自环时能正确工作。然后再调试与 DSP 接口是否正确，方法就是再通过 DSP 收发进行自环。调试模数转换器与 DSP 接口的程序如例 12.2 所示，该程序可在内部 DSP RAM 中运行。

【例 12.2】PCM Codec 通过 TMS320C203 的自环程序

```
;定义和矢量表同例 12.1
_main:
        DINT                ; 关中断
        SSXM                ; 置符号扩展方式
        RTXM                ; TXM＝0，外同步
        CNFD                ; B0 块为数据区
        LDPK    0           ; 数据页 0
        LACK    #08H        ; ACC＝08H
        SACL    IMR         ; 开串行口接收中断
        EINT                ; 开中断
LOOP:   B       LOOP

RINT_START:
        SST     #0,TST0     ; 保存 ST0
        SST1    #1,TST1     ; 保存 ST1
        LDPK    0           ; DP＝0
        SACH    TACCH       ; 保存 ACC 高 16 位
        SACL    TACCL       ; 保存 ACC 低 16 位
        LAC     DRR         ; 从串行口取数据
        SACL    DXR         ; 送数据至串行口
        ZALS    TACCL       ; 恢复 ACC 低 16 位
        ADDH    TACCH       ; 恢复 ACC 高 16 位
        LST     #1,TST1     ; 恢复 ST1
        LST     #0,TST0     ; 恢复 ST0
        EINT
        RET
```

12.2.5　软件调试

软件调试可基于 TMS320C2xx 仿真器进行，TMS320C2xx 仿真器具有 JTAG 标准仿真接口，支持 C 语言和汇编语言的混合源码调试，人机界面友好，使用比较方便。

软件调试采用分步骤调试的方法进行。首先是调试各个软件模块，然后进行系统软件联调。TMS320C203 软件模块包括：

（1）A/D 和 D/A 串行中断程序模块；

（2）闪速存储器读、写和擦除程序模块；

（3）与 89C51 通信的 INT0 中断程序模块；

（4）语音压缩程序模块；

（5）语音解压缩程序模块。

检验各程序模块正确性的方法需根据各模块的实际情况而定。例如，调试 A/D、D/A 串行中断程序模块的方法就是在中断程序中将 A/D 变换后得到的语音数据直接进行 D/A 变换，这样比较输入和输出的声音就可以判断中断程序的正确与否。闪速存储器程序模块的调试方法是比较写入和读出的对应数据。与 AT89C51 通信程序模块的调试要与 AT89C51 程序配合进行，调试的方法就是比较 TMS320C203 收到的数据与 AT89C51 发送的数据。语音压缩和解压缩程序模块是系统软件的核心，检验实时语音压缩和解压缩程序的方法是比较实时程序运行结果和用 C 语言定点模拟的结果。一般而言，当定点模拟程序与实时 DSP 程序在输入完全一致的条件下，其输出也应完全一致。调试软件模块时需要注意两点：一是注意要保证程序模块在被其他程序调用时运行正确且互相不破坏运行环境；二是注意要保证程序能在各种情况下都能正确运行，特别是要注意程序中各个变量在极限取值的条件下也能正常运行，这一点往往被忽视，从而导致在程序联调时出现问题。

程序的各个模块调试通过后，就可以进行系统程序的联调。系统程序联调实际上就是按照软件设计要求将各个程序模块有机地结合起来。本系统主程序中的命令分析和处理程序完全用 C 语言编制，因此，系统的程序联调比较方便。在各个程序模块调试正确的基础上，程序联调实际上就是保证各个程序模块之间以及与主模块之间的软件接口正确。在实际调试过程中遇到最多的问题就是各模块之间的接口问题。正确的软件接口就是各个模块之间不改变相互的运行环境，调用程序不破坏被调用程序的运行环境，被调用程序同样也不破坏调用程序的运行环境。软件接口正确后，联调就很容易通过。

12.2.6　独立系统实现

硬件和软件调试成功后，就可以将整个系统的软件从开发系统移植到独立的硬件系统上。将各个程序编译链接形成最后的可执行程序的过程如下。

（1）编译、汇编 C 程序和汇编程序，形成扩展名为 .obj 的目标文件。

对 C 程序的编译方法为（以 encode.c 为例）：

```
dspcl    encode.c
```

对 ASM 程序的汇编方法为（以 adda.asm 为例）：

```
dspa     adda.asm
```

（2）链接目标文件，形成扩展名为 .out 的输出文件。

首先编写一个链接命令文件，设该命令文件的文件名为 szly.cmd，则链接的方法为：

```
dsplnk    szly.cmd
```

（3）将.out 文件格式转换为适合于 EPROM 写入器的文件格式，连同 BOOT 信息，形成用于烧制 EPROM 的文件 szly.bin。

（4）将 szly.bin 文件写入 27C256 EPROM 中。

（5）将 EPROM 插入电路板的相应部分，系统即成为可独立运行的 DSP 系统。

12.3 一个基于 TMS320C31 DSP 系统的开发

TMS320C31 是 TI 公司生产的 DSP 芯片中一种性能价格比较高的浮点芯片，也是目前国内应用比较广泛的浮点 DSP 芯片之一。本节介绍一种基于 TMS320C31 的应用系统——多速率声码器，介绍该系统的软硬件设计和系统开发过程。

12.3.1 系统简介

多速率声码器是一种可工作于多种速率的语音压缩编码系统。该系统采用高效的多带激励（MBE）语音压缩编码算法，将语音的数码率从 PCM 编码的 64 kbit/s 降低到 1.2~8.0 kbit/s，并且保持相当的语音质量，即使在速率低至 1.2 kbit/s，语音仍保持相当的自然度，基本上可以识别熟人讲话。可广泛应用于有线与无线的窄带数字保密通信、卫星通信、移动通信、语音存储、语音邮件等应用场合，也可应用在数据网中实现语音通信。

本系统由 AT89C51 单片机和 TMS320C31 浮点 DSP 芯片构成。其中，AT89C51 作为整个系统的主机，完成系统的控制和通信功能，TMS320C31 作为系统的核心完成语音的压缩和解压缩功能。系统提供灵活的模拟话音接口和标准的 V.24 或 RS-232C 数字接口，提供 1.2/2.4/4.8/8.0 kbit/s 四种速率，可以根据外部提供的同步时钟自适应地改变工作速率，接口比较方便。

12.3.2 系统构成

图 12-4 是本系统的硬件构成框图，系统主要由 TMS320C31、AT89C51、模数转换器及 EPROM 和高速 RAM 等构成。图中，A/D 和 D/A 完成语音信号的模数和数模转换功能。EPROM 用于存放程序和已初始化的数据，RAM 用于存放实时运行的程序和数据，TMS320C31 实现语音编解码算法，而 AT89C51 则完成外部通信、声码器同步和与 TMS320C31 交换数据。系统的工作过程如下：系统加电后，AT89C51 复位后控制 TMS320C31 复位。TMS320C31 复位后，由内部固化的自引导程序（BOOT）将存于 EPROM 的程序和数据搬移至高速 RAM，然后 TMS320C31 开始运行声码器算法。声码器每 20ms 一帧运行一次编解码算法，与 AT89C51 交换一次数据。一方面 TMS320C31 将语音压缩后得到的数据传送至 AT89C51，由 AT89C51 发送至信道。另一方面，AT89C51 将从信道接收到的一帧数据传送至 TMS320C31，由 TMS320C31 还原为合成语音。由于 TMS320C31 的运算速度足以在一帧内完成语音的压缩和解压缩算法，因此，系统可以按全双工方式工作。

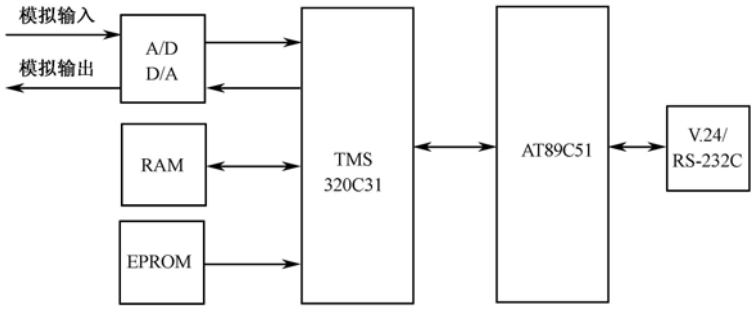


图 12-4　系统构成框图

12.3.3　系统软硬件设计

声码器作为数字通信系统中一个关键的终端设备，通常对其要求是体积尽量小、成本和功耗尽量低、可靠性高。因此，在设计声码器硬件时所遵循的原则就是在不影响系统性能的前提下尽量简化硬件设计。

1．TMS320C31 处理系统

TMS320C31 处理系统由 TMS320C31、EPROM、RAM、译码电路、模数转换电路等构成。其中，EPROM 为一片 27C512，构成 32 K×8 的存储空间，用于存储程序及初始化数据。RAM 为四片 7Y7C199-20，构成 32 K×32 的存储空间，程序就在这 32 K 字的空间运行。译码电路为一片 74F138，用于将 EPROM 和 RAM 映射至指定的地址空间，本系统采用 INT1 Boot 方式，EPROM 的地址为 0X400000~0X40FFFF，RAM 的地址空间为 0X10000~0X17FFF。

模数转换电路仍然采用 PCM 编译码器 MC14LC5480，为简化硬件，我们充分利用了 TMS320C31 提供的串行接口，使得 MC14LC5480 与 TMS320C31 能够以无硬件方法连接。TMS320C31 提供的串行接口可以用软件设置为多种工作方式，串行时钟既可设置为由外部提供，也可设置为由内部提供，有效电平可设置为高有效或低有效，这种灵活的软件设置方式给硬件设计带来了极大的便利。根据 MC14LC5480 的接口要求，将 TMS320C31 的串行口作如下设置：串行接口时钟设置为内时钟方式，由 TMS320C31 内部提供；用定时器 0 产生 8kHz 同步时钟；由串行 R/X 定时器产生收发时钟。下面我们先来计算定时器和串行口的有关时常数：

（1）用定时器 0 产生 8 kHz 同步时钟：f_{osc} =32.76 8MHz 时（TMS320C31-33），采用 CLOCK 方式，因此定时器产生的方波频率为

$$f=（f_{osc}/4）/（2\times \text{period_0}）$$

已知 f =8 kHz，可得定时器 0 的时常数为 period_0=512。

f_{osc}= 40.96 MHz 时（TMS320C31-40），采用 PULSE 方式，因此定时器产生的脉冲频率为

$$f=（f_{osc}/4）/（\text{period_0}）$$

已知 f =8 kHz，可得定时器 0 的时常数为 period_0=1280。

（2）串行 R/X 定时器产生收发时钟：f_{osc} =32.768 MHz 时（TMS320C31-33），采用 CLOCK 方式，f =（f_{osc} /4）/（2×period_rx），设收发时钟为 2.048 MHz，则有 period_rx=2。

f_{osc}= 40.96 MHz 时（TMS320C31-40），采用 PULSE 方式，f =（f_{osc}/4）/（period_rx），设收发时钟为 2.048 MHz，则有 period_rx=5。

【例 12.3】定时器和串行口设置程序（以 f_{osc}=32.768 MHz 为例）：

```
T0_CTLADDR      .word   000808020H          ; 定时器 0
T0_CTLINIT      .word   0000003C0H
SP0_ADDR        .word   000808040H          ; 串行口全局控制寄存器
RESET_SP        .word   02800300H
SP0_GLOBAL      .word   00E0003C4H
PERIOD_0        .word   512
PERIOD_RX       .word   00020002H           ; CLKX=CLKR=2.048 MHz

SP0_SET:
        LDI     @T0_CTLADDR,AR1
        LDI     0,R1
        STI     R1,*+AR1（4）
        LDI     @PERIOD_0,R1                ; 产生 8 kHz
        STI     R1,*+AR1（8 ）
        LDI     @T0_CTLINIT,R1
        STI     R1,*AR1                     ; 定时器 0 控制
        LDI     @SP0_ADDR,AR0               ; 初始化串行口
        LDI     111H,R1
        STI     R1,*+AR0（2）
        STI     R1,*+AR0（3）
        LDI     @SP0_GLOBAL,R1
        STI     R1,*AR0
        LDI     @PERIOD_RX,R1               ; R/X 定时器周期常数
        STI     R1,*+AR0（6）
        LDI     100H,R1
        STI     R1,*+AR0（5）               ; R/X 定时器计数器
        LDI     3CFH,R1
        STI     R1,*+AR0（4）               ; R/X 定时器控制
        RETS
```

TMS320C31 的软件主要包括初始化程序、主程序、串行口中断程序、定时器中断程序和与 AT89C51 通信程序。初始化程序完成 TMS320C31 的初始化，包括总线设置、中断矢量设置、定时器初始化、串行口初始化等。主程序完成声码器算法，包括语音分析、编码、译码、合成等。串行中断服务程序完成串行口语音的接收，定时器中断服务程序将合成语音发送至串行口。通信程序每隔一帧时间与 AT89C51 交换一次数据，TMS320C31 将一帧编码数据发送至 AT89C51，同时从 AT89C51 取回一帧数据。

2．AT89C51 与TMS320C31 的接口

为进一步简化硬件，我们在设计 AT89C51 与 TMS320C31 之间的接口电路时，没有采用通常的并行接口，而是用软件开发了一个串行接口，从而使两个处理器之间的硬件接口达到最简单。AT89C51 和 TMS320C31 的接口电路如图 12-5 所示。

两个处理器之间的通信过程如下：

（1）TMS320C31 向 AT89C51 发送数据：首先将要发送的一位数据置于 XF0，然后在 TCLK1 引脚上发一个低电平脉冲，向 AT89C51 发 INT0 中断，AT89C51 响应 INT0 中断，在中断服务

程序中读取 P1.0 引脚上的状态，完成一次发送过程。由于 TMS320C31 是按帧发送的，因此，为使 AT89C51 能正确接收，TMS320C31 在发送数据的第一比特时，在 TCLK0 引脚上产生一个高电平脉冲作为同步信号，而在发送后续比特时，TCLK0 引脚都为低电平。这样，AT89C51 在接收一位数据的同时判断 P1.2 引脚的状态，若 P1.2 为高，表明是第 1 比特，将接收计数器置为 1，从而保证一帧数据的正确接收。

（2）TMS320C31 接收 AT89C51 的数据：TMS320C31 向 AT89C51 发送数据和从 AT89C51 接收数据是以交替方式进行的，即发送完 1 位数据后接着接收 1 位数据，这样可以简化双方的通信过程。AT89C51 向 TMS320C31 发送数据时，首先将需要发送的数据位置于 P1.1，然后在 P1.4 产生一个低电平脉冲向 TMS320C31 发送 INT0 中断，TMS320C31 在 INT0 中断服务程序中读取 XF1 引脚的状态，1 位数据接收完毕。

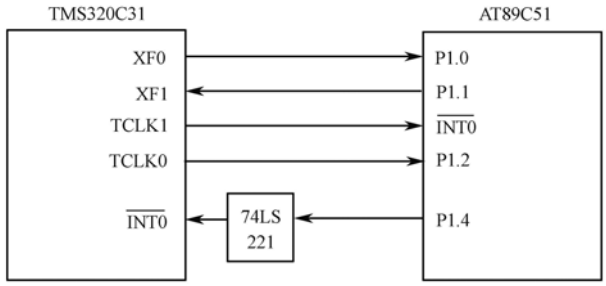


图 12-5　TMS320C31 与 AT89C51 的通信接口

3．AT89C51 与外部的通信

AT89C51 除了用于系统控制和与 TMS320C31 交换数据外，还用于与外部进行通信。声码器与外部通信一般采用标准的 V.24 或 RS-232C 接口。V.24 接口采用 TTL 接口电平，可直接由 AT89C51 软件实现。V.24 接口是一个同步串行接口，串行数据收发都是在相应的同步时钟作用下进行的。前面我们已经介绍过类似的接口，这里就不再重复了。在与外部通信程序中，AT89C51 除了发送和接收数据之外，还必须在接收数据中找出每一帧的起始位，即实现帧同步功能。

12.3.4　软硬件调试

TMS320C31 本身提供了用于仿真的 JTAG 仿真信号线。因此基于 TMS320C31 的 DSP 系统的硬件软件都可以用 TMS320C31 的仿真器方便地进行调试。

为了使 TMS320C31 的仿真器能够调试用户的硬件系统，在设计电路板时必须设计与仿真头相匹配的 12 根仿真线。利用这些仿真线，仿真器就可以对用户系统进行硬件软件调试。在调试通过用户系统的硬件后，软件就可以直接在用户系统上进行调试。

1．系统硬件调试

设计 TMS320C31 硬件系统时也应遵循 12.2.4 节所介绍的一些基本的设计原则。此外，为了保证系统运行的可靠性，减少干扰，印制电路板一般应采用四层板以上的工艺。例如，采用四层板工艺时，上下两层可作为布线层，而中间两层可分别作为电源层和地线层。

系统调试之前应对电路板作常规检查，在确保电源等关键的接线正确之后就可以接上仿真头、插上芯片加电调试。当然调试电路板前首先应确保所用芯片能够正常工作。如果 TMS320C31 各电源引脚的电压正常，晶体振荡器正常工作，仿真线连接正确，则运行仿真器

的调试软件后就可在 PC 的屏幕上显示调试器界面。否则应检查仿真线连接是否正确、TMS320C31 电源引脚电压是否正常以及晶体振荡器是否正常工作。如果调试器程序运行正常，则说明 TMS320C31 已经正常工作。此时，可以用调试器检查片内 RAM，例如要查看 0x809800 地址中的内容，输入 mem 0x809800 命令并回车后即可在存储器窗口中观察到该地址的内容。若要修改该地址的内容，只需用鼠标点亮原来的数据，输入所需数值并回车即可。

在 TMS320C31 正常工作之后，接下来就可以调试外部 RAM 了。用仿真器调试外部 RAM 也比较方便，只要对 RAM 进行修改和读操作就可以判断 TMS320C31 读写外部 RAM 是否正常。本系统中外部 32 K×32 的 RAM 映射至 0x10000~0x17FFF 的地址空间。用仿真器提供的填充 RAM 功能将数据写入到外部 RAM，再查看 RAM 的内容就能判断 RAM 与 TMS320C31 接口是否正常。如果读出的内容与填充值完全相同，表明 TMS320C31 对 RAM 的读写完全正确。如果读出的内容呈现有规律性的错误，如低 8 位不对，则可将故障压缩至负责低 8 位数据的那片存储器上。如果完全不对，则应仔细检查系统的译码电路、地址线、数据线、读写线等连接是否正确。当然也不排斥某些非常特殊的情况，即对 TMS320C31 的片内 RAM 读写完全正确，与外部 RAM 的连接也完全正确，就是对外部 RAM 不能正确读写，作者就遇到过类似情况。经过反复检查确认了 TMS320C31 的有些地址线、数据线引脚与内部信号线断路，可能是由于贴片时温度过高所至。

系统中还有一片 EPROM，存储程序及 Boot 所需的一些数据。调试 EPROM 接口也可以用仿真器的存储器查看功能。本系统中 EPROM 映射至 0X400000~0X40FFFF 的地址空间。设 EPROM 的前面四个存储单元的内容为 0X11、0X22、0X33 和 0X44，输入 mem 0X400000 及回车，应在存储器窗口中看到如下结果：

400000　　xxxxxx11 xxxxxx22 xxxxxx33 xxxxxx44

由于 EPROM 为 8 位，因此前面 24 位数据是无效数据。EPROM 接口正确后就可以按照第 6 章介绍的方法将 Boot 数据和用户程序写入 EPROM。将写好的 EPROM 插上电路板，加电后再插上仿真头，运行仿真程序并在存储器窗口中查看用户程序是否已经正确地被引导到目的地址。

上述硬件调试成功后，就可以直接用调试成功的硬件系统来调试软件、其他硬件（模数转换芯片）和系统软件。模数转换芯片与 TMS320C31 的接口可以用软件自环的方法进行调试，TMS320C31 在串行接收中断中接收一个数据并立即将它发送出去。下面是 TMS320C31 接收中断程序。

【例 12.4】PCM Codec 自环测试：串行口接收中断服务程序

```
IO_ADD    .word     0x808000
RECEIVE0:
          PUSH      ST
          PUSH      AR0
          PUSH      R0
          LDI       @IO_ADD,AR0
          LDI       *+AR0（4CH）,R0        ;取数
          STI       R0,*+AR0（48H）        ;送数
          POP       R0
          POP       AR0
          POP       ST
          RETI
```

2．系统软件调试

TMS320C31 的软件主要是在仿真器上调试的。为了既提高程序效率，又提高编程效率，我们在编写整个 TMS320C31 系统软件时使采用三种不同方法，即有些程序模块直接用汇编语言编写，有些程序模块直接用 C 语言编写，而有些程序模块则用 C 和汇编的混合语言编写。完全用汇编语言编写的模块包括串行口中断程序、定时器中断程序、FFT 程序、滤波程序等，用汇编语言编写这些模块可使程序的效率达到最高。完全用 C 语言编写的程序主要是一些不影响系统性能的程序模块，如初始化程序中的一些子模块。主程序则采用 C 和汇编的混合语言编写而成，采用混合语言编写方法，在程序运算量大的地方用手工编写汇编语言，其他地方直接用 C 语言编写，这样做既可以提高程序效率，也可以利用 C 程序的结构及仿真器的 C 源码调试功能，提高调试效率。

仿真器支持 C 语言调试，为系统软件调试提供了极大的便利。我们模拟语音编码算法采用的是 C 语言，这个用 C 语言编写的模拟程序稍作修改就可以在仿真器上运行。虽然由于运算量大无法使算法实时运行，但是可以进行实时模拟。也就是说，给 C 程序输入合适的数据，TMS320C31 就可以立即计算出输出数据。在输入数据和其他输入条件完全一致的条件下，输出数据应该和 C 模拟得出的数据完全一致。为了找出程序中哪些程序段运行时间长，以便将它们更改为汇编程序，可以利用调试器的 RUNB 命令计算出每一段程序所需的运算时间。

调试器可以方便地查看 C 程序中的任意变量或数组，而要查看汇编变量则不太方便。因此，在对 C 程序改为汇编程序时，为了便于调试，不妨可以将每段汇编程序的结果重新写入 C 变量或数组，待程序调试通过后再删去。实践证明，这种方法可以大大提高程序调试效率。

AT89C51 软件全部用汇编语言编写并在相应的开发装置上调试通过。



12.3.5　独立系统实现

软硬件调试成功后，就可以脱离仿真器。将整个系统的软件写入 EPROM，插入电路板，就可使系统成为独立运行的 DSP 系统。将各个程序编译或汇编并链接形成最后的可写入 EPROM 文件的过程如下：

（1）编译汇编 C 程序和汇编程序，形成扩展名为 .obj 的目标文件。对 C 程序的编译方法为（以 mbefxhc.c 为例）：

```
cl30    -v30    mbefxhc.c
```

对 asm 程序的汇编方法为（以 main31.asm 为例）：

```
asm30 -v30   main31.asm
```

（2）链接目标文件，形成扩展名为 .out 的输出文件。首先编写一个链接命令文件，设文件名为 mbe.cmd，则链接的方法为：

```
lnk30   mbe.cmd
```

（3）将.out 文件格式转换为 ti-tagged 文件格式，形成 ti 格式文件 mbe.ti。

```
rom30   -t    mbe.out
```

（4）将 mbe.ti 文件再转换为 8 位字节宽度的二进制文件 mbe.bin，转换方法为：

```
tb31    mbe
```

上述命令行中 TB31 是作者编写的一个应用程序，它将 TI 格式的文件转换为具有 Boot 功能的二进制文件，适合于 8 位字节宽度的 EPROM Boot 方式，其 C 语言源程序见附录。若该

程序不能适合于读者的应用需要，通过在这个程序的基础上作适当的修改即可。

（5）将 mbe.bin 文件用 EPROM 写入器写入 27C512 EPROM 中。

（6）将 EPROM 插入电路板的 EPROM 位置，系统即成为可独立运行的 DSP 系统。

本系统采用的链接命令文件见例 12.5。

【例 12.5】TMS320C31 系统链接命令文件

```
        /*文件名：mbe.cmd*/
main31.obj          /*初始化程序*/
mbefxhc.obj         /*分析合成程序*/
mbe_en.obj          /*编码程序*/
mbe_de.obj          /*解码程序*/
fft_sub.obj         /*FFT 程序*/
com_sub.obj         /*通信程序*/
-c                  /*ROM 初始化*/
-o mbe.out          /*生成 mbe.out*/
-m mbe.map          /*生成 mbe.map*/
-l rts30.lib        /*链入 rts30.lib*/

MEMORY
{
        EXT_RAM :   org = 0x010000, len = 0x17FFF     /*外部 32K 高速 RAM*/
        RAM0:       org = 0x809800, len= 0x400,       /*内部 RAM 块 0*/
}
SECTIONS
{
          .text :    {} > EXT_RAM
          .data :    {} > EXT_RAM
          .cinit:    {} > EXT_RAM
          .bss:      {} > EXT_RAM
          .stack:    {} > RAM0
}
```

12.4 一个基于 TMS320VC5409 DSP 应用系统的开发

TMS320VC5409 是一种高性能的定点 DSP 芯片，目前已经在很多领域得到广泛的应用。本节介绍一个采用 TMS320VC5409 实时实现 G.729A 8 kbit/s 国际标准语音压缩算法的 DSP 实时系统。首先简要介绍一下 G.729A 语音编码算法，然后着重介绍这个系统的软硬件设计、调试及开发过程。

12.4.1 G.729A 及系统简介

G.729 是国际电信联盟 ITU-T 制定的工作速率为 8kbit/s 的语音压缩算法，该算法采用“共轭结构-代数码激励线性预测”算法，简称 CS-ACELP 算法。该标准于 1995 年 11 月通过，主要应用在个人移动通信、数字卫星系统和高质量数字移动无线通信等领域。为了降低复杂度，ITU-T 在

1996 年 5 月通过了 G.729 的附件 A，即 G.729A。1996 年 6 月，ITU-T 又制定了 G.729 的附件 B，即 G.729B，使它能用于 ITU-T V.70 终端。G.729B 在 G.729A 的基础上，使用了 VAD/DTX/CNG 算法，即话音激活检测/不连续传输/舒适噪声产生算法。G.729、G.729A 的编码器属性如表 12-1 所示。G.729 和 G.729A 两级级联语音质量优于 32kbit/s G.726 四级级联语音质量；无噪声时，G.729A 稍差于 G.729，基本等价于 32kbit/s 的 G.726；在音乐噪声条件下，G.729 和 G.729A 都比 G.726 差，G.729 和 G.729A 基本等价；在随机比特误码（10^{-3}）和随机帧丢失（3%）条件下，无噪声时，G.729 和 G.729A 以及两者互联时，未发现明显差别；而在有噪声（办公室）时，噪声掩盖了帧丢失的影响，G.729、G.729A 和它们的互联未发现明显差别，与 G.726 等价。对于 G.729B，主观测试结果表明：无论是有噪声还是无噪声，G.729B 的性能都等价于 G.729A 的性能。

本系统由 TMS320VC5409 定点 DSP 芯片、Flash、模数转换芯片、双口 RAM 等构成。其中，TMS320VC5409 作为整个系统的核心，完成包括语音压缩和解压缩在内的所有软件功能。

表 12-1　G.729 和 G.729A 的编码属性比较

参　　数	G.729	G.729A
比特率/（kbit/s）	8	8
帧　长/ms	10	10
子帧长/ms	5	5
运算时延/ms	15	15
运算量/MIPS	20	10.5

12.4.2　系统构成

图 12-6 是本系统的硬件构成框图，系统主要由 TMS320VC5409、模数转换器及 Flash 和双口 RAM 等构成。图中，A/D 和 D/A 完成语音信号的模数和数模转换功能。Flash 用于存放程序和已初始化的数据，双口 RAM 用于与外部交换语音压缩数据。系统的工作过程如下：系统加电 TMS320VC5409 复位后，由其内部固化的自引导程序（BOOT）将存于 Flash 的程序和数据搬移至内部 RAM，然后 TMS320VC5409 开始运行语音编码算法。DSP 每 10 ms 运行一次编解码算法，并与双口 RAM 交换一次数据。一方面 TMS320VC5409 将语音压缩后得到的数据写入双口 RAM，由外部系统读出并送至信道。另一方面，外部系统将对方的编码数据送到双口 RAM，由 TMS320VC5409 从双口 RAM 中读出并还原为合成语音。由于 TMS320VC5409 的运算速度足以在 10ms 内完成语音的压缩和解压缩算法，因此，系统可以按全双工方式工作。

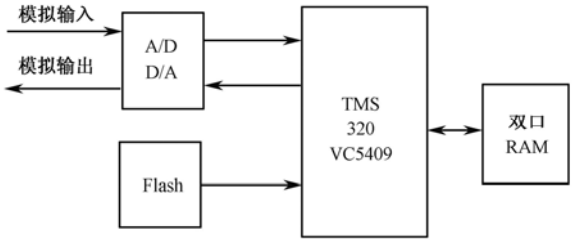


图 12-6　TMS320VC5409 DSP 系统构成框图

12.4.3 系统软硬件设计

本系统由 TMS320VC5409、Flash、双口 RAM、时序产生电路、模数转换电路及电平缓冲接口芯片等构成。其中，Flash 为一片 29LV020，构成 256 K×8 的存储空间，用于存储程序及初始化数据；双口 RAM 为一片 7Y7C135-55，构成 4K×8 的存储空间，用于与外部交换数据；时序产生电路用于产生系统所需的各种时序；模数转换电路由 MC14LC5480 及其外围电路构成；电平缓冲接口芯片采用 74LVC16245。为简化硬件，本系统不用译码电路。Flash 既映射在数据空间，又映射在程序空间。双口 RAM 映射在数据空间。双口 RAM 与 Flash 占用的数据空间地址是重叠的，但 Flash 使用 DSP 外部数据总线的低 8 位，而双口 RAM 则使用 DSP 外部数据总线的高 8 位，因而彼此并不冲突。本系统采用外部并行 8 位 BOOT 方式。Flash 的数据空间地址为 0000H~FFFFH，程序空间地址为 0000H~FFFFH、010000H~01FFFFH、020000H~02FFFFH、030000H~03FFFFH。双口 RAM 的数据空间地址为 8000H~FFFFH。

1．电源设计

本系统采用 TMS320VC5409 DSP 芯片，DSP 的内核电压和接口电压分别为 1.8 V 和 3.3 V，因此，本系统电压有 5 V、3.3 V 和 1.8 V 三种。其中，双口 RAM、模数转换电路和时序产生电路均由 5 V 供电，Flash 和电平缓冲接口芯片由 3.3 V 供电。本系统的 5 V 由外部供给，3.3 V 和 1.8 V 由系统内部电压调节器提供，该电压调节器采用 TPS73HD301 实现。

2．DSP设计

DSP 设计主要考虑了以下几个方面：

（1）时钟电路。采用外接晶振，内部时钟使能方式。设置 CLKMD1=1，CLKMD2=0，CLKMD3=1。加电时，DSP 的工作时钟为外部晶振频率的 1/4。

（2）串行接口。TMS320VC5409 提供了三个具有缓冲功能的串行口。本系统采用标准串行口方式。

（3）复位电路。采用简单的阻容复位与外部复位相结合的方式。系统既可以直接采用简单的阻容复位方式，同时，外部也可以随时对 DSP 进行复位，系统受外部控制。

（4）电源考虑。本系统采用 TPS73HD301 电源调节器提供 3.3 V 和 1.8 V，其中 3.3 V 电压是固定的输出，而提供 1.8 V 电压的一路是可调节的，这使系统更加灵活。

（5）外接存储器地址及数据分配。为简化设计，本系统不设计译码电路，Flash 既映射在程序区，又映射在数据区，而双口 RAM 映射在数据区。为使 Flash 和双口 RAM 的数据不冲突，Flash 采用外部数据的低 8 位，双口 RAM 采用外部数据的高 8 位。

3．Flash接口设计

设计 Flash 与 DSP 的接口时，主要考虑如下几点：

（1）采用 DSP 的外部 8 位 BOOT 方式；

（2）通过 DSP 的仿真系统，能够将程序和数据写入到 Flash 中；

（3）系统独立运行时，能够从 Flash 中读出程序装入到内部 RAM 中；

（4）接口尽量简单。

根据以上几点，我们设计的 Flash 接口如图 12-7 所示。

4．双口RAM设计

双口 RAM 映射在 DSP 的外部数据区，因此双口 RAM 的外部地址映射在 DSP 数据区的

8000H~FFFFH 范围。双口 RAM 对应于 DSP 侧的接口如图 12-8 所示。

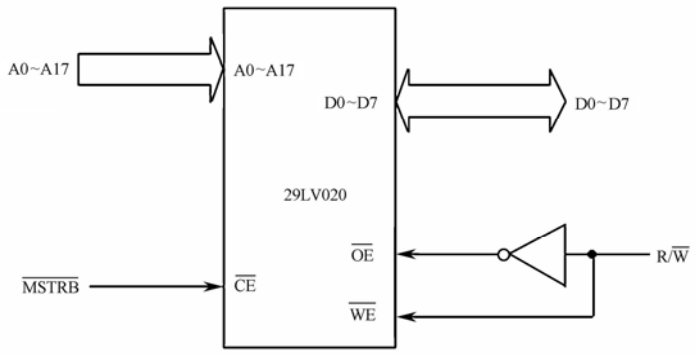


图 12-7　DSP 与 Flash 的接口

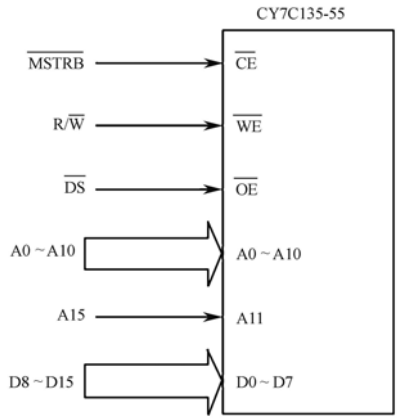


图 12-8　双口 RAM 与 DSP 的接口

12.4.4　系统调试

整个系统的调试包括 3 个部分：硬件调试、软件调试和总体调试，这三部分调试都可借助于 TMS320VC5409 的仿真器进行调试。

为了使 TMS320VC5409 的仿真器能够调试用户系统，在设计电路板时必须设计与仿真头相匹配的 14 根仿真线（JTAG）。利用 JTAG 仿真头，仿真器就可以对用户系统进行硬件软件调试。首先调试硬件系统，调试通过后，软件就直接可以在用户硬件系统上进行调试了。

1．硬件调试

硬件焊接完成后，首先测试一下电压是否正确。然后，可用示波器或逻辑分析仪测试一下 DSP 的 CLKOUT 引脚是否有信号输出，并测试一下该信号的频率是多少，对照设定的 CLKMD1、CLKMD2、CLKMD3，检查频率是否正确。如果 CLKOUT 的输出正确，则一般来说，DSP 部分的硬件基本正常。

之后，可以将仿真系统与硬件系统相连接。开机加电并进入运行仿真系统软件，如果仿真系统的软件能够正常运行，则至少 DSP 内部就能够正常运行了。

接下来可以调试 DSP 外围硬件了。与 Flash 的连接部分需要编写相应的程序来调试，例如 Flash 的擦除、编程及读取等。如果能够将写入的数据正确地读出，则表明 Flash 的接口是正确的。与双口 RAM 的接口比较简单，调试也比较方便。只需要用仿真器的存储器观察命令（“mem”）就可以了，在窗口中能够任意改变数值表明双口 RAM 能读能写。

模数接口的调试涉及一些软件。只要将串行口接收中断打开，并在串行口接收中断程序中，将收到的样值立即送回串行口，在模拟输入端送一个正弦信号（300~3400Hz），观察模拟输出端，如果在输出端能够得到与输入端相同的信号，则表明模拟转换电路及其与 DSP 的接口是正确的。

2．软件调试

TMS320VC5409 的软件主要是在仿真器上调试的。同样，为了既提高程序效率，又提高编程效率，我们在编写整个 TMS320VC5409 系统软件时采用三种不同方法，有些程序模块直接用汇编语言编写，有些程序模块直接用 C 语言编写，而有些程序模块则用 C 和汇编的混合

语言编写。

为了提高程序效率，将定点 C 程序用手工的方法编写汇编语言是十分必要的。编写汇编语言程序时，首先可以将 C 程序用 C 编译器进行编译，得到汇编程序，然后在该汇编程序的基础上，保留入口和出口部分，将核心部分进行手工编写，这样可以避免入口和出口参数传递发生错误。当然，读者采用这种方法时，首先必须十分清楚参数的传递方法，否则，一旦发生错误将无从下手。

调试软件时，通过比较 C 程序的模拟结果与汇编程序的结果来实现。一般来说，汇编语言实现的结果应与 C 程序模拟的结果完全一致。

3．总体调试

总体调试包括系统的初始化、软硬件的联合调试等。系统初始化设置系统的工作环境，因此极为重要。本系统的初始化工作主要包括：

（1）中断矢量的重定位。将中断矢量表的起始地址设置在 7F00H 处，设置方法是对 PMST 寄存器进行设置。

（2）工作时钟设置。系统开始时，工作时钟是根据外部三个引脚的设置进行的。例如，本系统 CLKMD1~3=101b，因此开始时的工作时钟为晶体频率的 1/4。如晶体的频率为 20 MHz，则开始的时钟为 5 MHz。正常工作时的频率为 100 MHz，因此需设置的倍数为 5（20×5=100）。设置的方法是对 CLKMD 寄存器进行设置。

（3）等待状态数的设置。外部双口 RAM 的速度是 55 ns，而正常工作的指令周期是 1/100MHz=10 ns，因此需插入 5 个等待状态。设置的方法是对 SWWSR 寄存器进行设置。

（4）中断设置。打开串行口和 INT0 中断（外部脉冲每 10 ms 中断一次）。设置方法是对 IMR 寄存器进行设置。

（5）其他设置。其他初始化设置包括串行口初始化、ST0、ST1 初始化等。

（6）软硬件联合调试。联合调试是将所有程序综合在一起利用仿真器和带仿真接口的用户硬件系统进行调试。程序包括：初始化程序、语音编解码程序、串行口中断服务程序、INT0 中断服务程序等。初始化程序完成系统的初始化，语音编解码程序完成 G.729A 算法，串行口中断服务程序完成语音样值的输入和输出，INT0 中断服务程序完成与双口 RAM 每 10 ms 的编码数据的交换。



12.4.5 独立系统形成

软硬件调试成功后，首先通过仿真器将用户 Boot 程序通过仿真器写入 Flash，然后在通过仿真器将整个系统的软件写入 Flash。两个软件写入 Flash 后，系统就可以成为独立运行的 DSP 系统。

将系统软件编译链接并写入到 Flash 中的过程如下：

（1）编译汇编 C 程序和汇编程序，形成扩展名为 .obj 的目标文件。对 C 程序的编译方法为（以 lpc.c 为例 ）：

```
cl500    lpc.c
```

对 asm 程序的汇编方法为（以 adda.asm 为例 ）：

```
asm500   adda.asm
```

（2）链接目标文件，形成扩展名为 .out 的输出文件。首先编写一个链接命令文件，设文件名为 coder8k.cmd，则链接的方法为：

```
lnk500    coder8k.cmd
```

（3）将.out 文件格式转换为 TI-tagged 文件格式，形成 TI 格式文件 coder8k.t0

```
hex500    -t     coder8k.out
```

（4）将 coder8k.t0 文件再转换为 ASCII 码文件 coder8k.asc，转换方法为：

```
ta54     coder8k
```

上述命令行中 ta54 是作者编写的一个应用程序，它将 TI 格式的文件转换为具有一定格式满足用户 Boot 程序需要的 ASCII 码文件。

（5）将 coder8k.asc 结合到写 Flash 的汇编语言文件 write.asm 中。

（6）对 write.asm 进行编译链接，形成 write.out。

（7）将 write.out 装入到仿真器中，运行该程序，将用户程序写入到 Flash。

（8）将系统脱离仿真器，系统即可独立运行。

本系统采用的链接命令文件见例 12.6。

【例 12.6】TMS320VC5409 系统链接命令文件

```
/*文件名：coder8k.cmd*/
-c
-o coder8k.out
-m coder8k.map

    acelp_ca.obj
de_acelp.obj
dec_gain.obj
dec_lag3.obj
dec_ld8a.obj
lspdec.obj
post_pro.obj
postfilt.obj
bits.obj
filter.obj
qua_gain.obj
cod_ld8a.obj
taming.obj
pitch_a.obj
pred_lt3.obj
lspgetq.obj
gainpred.obj
lpc.obj
cor_func.obj
pre_proc.obj
adda.obj
decoder.obj
p_parity.obj
qua_lsp.obj
lpcfunc.obj
oper_32b.obj
```

```
basic_op.obj
tab_ld8a.obj
dspfunc.obj
util.obj
sbbs.obj

-l     rts.lib
-stack    0x800

MEMORY
{
   PAGE 0:    PROG:    origin = 0xa00，length 0x7500              /*程序和数据区*/
              VECT:    origin = 0x7f00，length 0x80               /*矢量表*/
              PAGE 1: STACK1: origin = 0x60，length 0x9a0         /*堆栈区*/
              offchip:  origin = 0x8000，length 0x800             /*外部双口 RAM*/
}

SECTIONS
{      .text         : {}   >PROG PAGE 0
       .cinit        : {}   >PROG PAGE 0
       .table        : {}   >PROG PAGE 0
       .bss          : {}   >PROG PAGE 0
       vectors       : {}   >VECT PAGE 0
       .stack        : {}   >STACK1 PAGE 1
       .block        : {}   >offchip PAGE 1
}
```

12.5 小　　结

本章以三个 DSP 应用系统为例，分别介绍了 TMS320C203、TMS320C31、TMS320VC5409 三种 DSP 芯片在实现应用系统时的软硬件设计、调试和有关开发方法。了解实际应用系统的开发过程对于灵活掌握 DSP 芯片的应用开发是非常有帮助的。对于其他 DSP 芯片的应用设计，读者可参考本章的三个示例并结合实际系统进行设计。

12.6 习题与思考题

（1）试用 TMS320C203 DSP 芯片设计一个 DSP 应用系统。系统应包括 32 K 的扩展 RAM（既可做程序区也可做数据区）、EPROM、模数转换，以及能和计算机接口的 UART 接口。

（2）试用 TMS320C31 DSP 芯片设计一个 DSP 应用系统。系统应包括 32 K 的扩展 RAM、EPROM 和模数转换。

（3）试用 TMS320VC5402 DSP 芯片设计一个 DSP 应用系统。系统应包括 EPROM、模数转换以及和 89C51 单片机相接的 HPI 接口。

第 13 章　数字滤波器算法的 DSP 实现

13.1　引　　言

在数字信号处理中，滤波占有极其重要的地位。数字滤波是语音和图像处理、模式识别、谱分析等应用中的一个基本处理算法。与模拟滤波相比，数字滤波具有很多突出的优点，例如它可以满足滤波器对幅度和相位特性的严格要求，可以避免模拟滤波器无法克服的电压漂移、温度漂移和噪声等问题。用 DSP 芯片实现数字滤波除了具有稳定性好、精确度高、不受环境影响等优点外，还具有灵活性好的特点。用可编程 DSP 芯片实现数字滤波可通过修改滤波器的参数十分方便地改变滤波器的特性。

本章讨论 FIR 和 IIR 滤波器以及自适应滤波器的 DSP 实现方法。在简要介绍滤波器的基本原理、结构和设计方法之后，通过实例介绍用定点和浮点 DSP 汇编语言高效实现数字滤波的方法。

13.2　FIR 滤波器的 DSP 实现

13.2.1　FIR 滤波器的基本原理和设计方法

13.2.1.1　FIR 滤波器的特点

对于许多应用来说，数字滤波器一般具有如下差分方程：

$$y(n)=\sum_{k=0}^{N-1}a_k x(n-k)+\sum_{k=0}^{N-1}b_k y(n-k) \tag{13-1}$$

式中，x（n）为输入序列，y（n）为输出序列，a_k 和 b_k 为滤波器系数，N 是滤波器的阶数。式（13-1）中，若所有的 b_k 均为零，则有：

$$y(n)=\sum_{k=0}^{N-1}a_k x(n-k) \tag{13-2}$$

式（13-2）就是 FIR 滤波器的差分方程。不失一般性，下面我们用

$$y(n)=\sum_{k=0}^{N-1}h(k)\cdot x(n-k) \tag{13-3}$$

来表示 FIR 滤波器的差分方程。对式（13-3）进行 Z 变换，整理后可得 FIR 滤波器的传递函数：

$$H(z)=\frac{Y(z)}{X(z)}=\sum_{k=0}^{N-1}h(k)z^{-k} \tag{13-4}$$

如图 13-1 所示是 FIR 滤波器的一般结构。

在数字滤波器中，FIR 滤波器具有如下几个优点：

（1）可以在幅度特性随意设计的同时，保证精确、严格的线性相位；

（2）由于 FIR 滤波器的单位脉冲响应 h（n）是有限长序列，因此 FIR 滤波器没有不稳定的问题；

（3）由于 FIR 滤波器一般为非递归结构，因此在有限精度运算下，不会出现递归型结构中的极限震荡等不稳定现象，误差较小；

（4）FIR 滤波器可采用 FFT 算法实现，从而提高运算效率。

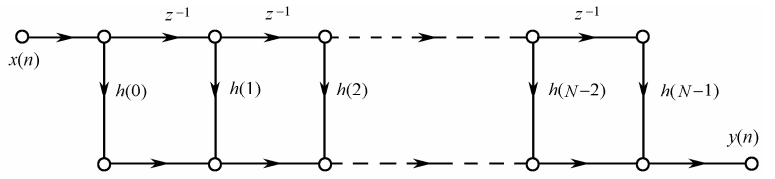


图 13-1　FIR 滤波器的一般结构

13.2.1.2　FIR 滤波器的设计方法

FIR 滤波器的设计方法主要有窗函数法和频率抽样设计法。其中窗函数法是基本的设计方法，下面我们简单地介绍一下这种方法。设如果所希望得到的滤波器的理想响应为 $H_{\mathrm{d}}(\mathrm{e}^{\mathrm{j}\omega})$，那么 FIR 滤波器的设计就在于寻找一个传递函数 $H(\mathrm{e}^{\mathrm{j}\omega})=\sum_{n=0}^{N-1}h(n)\mathrm{e}^{-\mathrm{j}n\omega}$ 去逼近 $H_{\mathrm{d}}(\mathrm{e}^{\mathrm{j}\omega})$，设

$$\begin{aligned}H_{\mathrm{d}}(\mathrm{e}^{\mathrm{j}\omega})&=\sum_{n=-\infty}^{\infty}h_{\mathrm{d}}(n)\mathrm{e}^{-\mathrm{j}n\omega}\\ h_{\mathrm{d}}(n)&=\frac{1}{2\pi}\int_{0}^{2\pi}H_{\mathrm{d}}(\mathrm{e}^{\mathrm{j}\omega})\mathrm{e}^{\mathrm{j}n\omega}\mathrm{d}\omega\end{aligned}\qquad(13\text{-}5)$$

这里 $h_{\mathrm{d}}(n)$ 是傅里叶级数的系数。在这种逼近中，最直接的一种方法就是从单位脉冲响应 h（n）入手，使 h（n）逼近理想的单位脉冲响应 $h_{\mathrm{d}}(n)$。由于 $h_{\mathrm{d}}(n)$ 是一个无限长序列，因此，最简单的方法就是对 $h_{\mathrm{d}}(n)$ 作截尾处理，即得到一个近似的传递函数：

$$H_{\mathrm{a}}(\mathrm{e}^{\mathrm{j}\omega})=\sum_{n=-Q}^{Q}h_{\mathrm{d}}(n)\mathrm{e}^{-\mathrm{j}n\omega}\qquad(13\text{-}6)$$

式（13-6）中，Q 就是最终确定 FIR 滤波器的阶数，Q 越大，近似程度就越高。对 $h_{\mathrm{d}}(n)$ 截尾，实际上就是对 $h_{\mathrm{d}}(n)$ 乘上一个矩形窗 $R_N(n)$，即

$$h(n)=h_{\mathrm{d}}(n)R_N(n)\qquad(13\text{-}7)$$

令 $z=\mathrm{e}^{\mathrm{j}\omega}$，则

$$H_{\mathrm{a}}(z)=\sum_{n=-Q}^{Q}h_{\mathrm{d}}(n)z^{-n}\qquad(13\text{-}8)$$

其脉冲响应系数为 $h_{\mathrm{d}}(-Q),h_{\mathrm{d}}(-Q+1),\cdots,h_{\mathrm{d}}(-1),h_{\mathrm{d}}(0),h_{\mathrm{d}}(1),\cdots,h_{\mathrm{d}}(Q-1),h_{\mathrm{d}}(Q)$。为使式（13-8）具有因果性，延迟 Q 个样值，可得：

$$H(z)=z^{-Q}H_{\mathrm{a}}(z)=\sum_{n=-Q}^{Q}h_{\mathrm{d}}(n)z^{-n-Q}\qquad(13\text{-}9)$$

令 $n+Q=k$，上式成为

$$H(z)=\sum_{k=0}^{2Q}h_{\mathrm{d}}(k-Q)z^{-k} \tag{13-10}$$

令 $h(k)=h_{\mathrm{d}}(k-Q)$，N=2Q，得

$$H(z)=\sum_{k=0}^{N}h(k)z^{-k} \tag{13-11}$$

式中，h（k）是脉冲响应系数，这里

$$h(0)=h_{\mathrm{d}}(-Q),h(1)=h_{\mathrm{d}}(1-Q),\cdots,h(Q)=h_{\mathrm{d}}(0),\cdots,h(2Q)=h_{\mathrm{d}}(Q)$$

脉冲响应系数具有对称特性，即 h（n）=h（-n）。例如，若 Q=9，则滤波器具有 2Q+1=19 个系数 h（0）, h（1）,…, h（18），这些系数关于 h（10）对称：

h（0）= h（18），h（1）= h（17），h（2）= h（16），… ，h（9）= h（11）　（13-12）

在设计 FIR 滤波器中，一个重要的计算就是加窗。采用矩形窗是最直接和简便的方法，但采用矩形窗存在较大的 Gibbis 效应，且矩形窗的第一旁瓣与主瓣相比仅衰减 13dB。因此实际设计中，一般采用其他窗函数，比较常用的窗函数有 Hanning 窗、Hamming 窗、Blackman 窗、Kaiser 窗等。采用 Hanning 窗使能量集中在主瓣内，第一旁瓣的衰减为 31dB，但主瓣的宽度增加了 1 倍。采用 Hamming 窗使 99.9%以上的能量集中在主瓣，第一旁瓣的衰减达 43dB，主瓣的宽度也是矩形窗的 2 倍。Blackman 窗进一步抑制旁瓣，使第一旁瓣的衰减达到 58dB，但主瓣的宽度是矩形的 3 倍。上述三种窗函数都是以增加主瓣宽度为代价换取一定程度的旁瓣抑制。而 Kasier 窗可以通过调整参数值来折中选择主瓣宽度和旁瓣衰减，采用 Kaiser 窗设计 FIR 滤波器具有很大的灵活性。

13.2.1.3　FIR 滤波器的设计工具

根据上面介绍的方法，可以编写计算机程序来设计 FIR 数字滤波器。许多数字信号处理的文献上一般都提供有关设计滤波器的高级语言程序等。有许多数字信号处理程序包也可直接用来设计数字滤波器。此外，采用著名的 MATLAB 程序包也可方便地设计数字滤波器。

MATLAB 是美国 Mathwork 公司于 1967 年推出的软件包，它是一种功能强、效率高、便于进行科学和工程计算的交互式软件包，其中包括：一般数值分析、矩阵运算、数字信号处理、建模和系统控制、优化等应用程序。MATLAB 中的工具箱（Toolbox）包含了许多实用程序。滤波器设计就包含在 Toolbox 下的 Signal 中。下面我们简要介绍一下用 MATLAB 设计 FIR 滤波器的方法。

在 MATLAB 中，提供了两种 FIR 滤波器设计方法。一种是窗函数设计法，即 FIR1；另一种是任意形状 FIR 滤波器设计法，即 FIR2。下面我们仅以 FIR1 为例说明其使用方法。

（1）设计低通 FIR 滤波器（采用 Hamming 窗）：

B=FIR1（N, Wn）

其中，N 为滤波器的阶数，Wn 为归一化截止频率，0.0<Wn<1.0，1.0 对应于半抽样速率。返回值 B 为一个（N+1）维矢量，即滤波器系数。

（2）设计高通 FIR 滤波器（采用 Hamming 窗）：

B=FIR1（N, Wn，’high’）

其中，N 和 Wn 的含义与低通滤波器相同。

（3）设计带通 FIR 滤波器（采用 Hamming 窗）：

B=FIR1（N, Wn）

其中，Wn 是一个二维矢量，Wn=[W1 W2]，W1 和 W2 分别对应于通带的低端频率和高端频率。

（4）设计带阻 FIR 滤波器（采用 Hamming 窗）：

B=FIR1（N, Wn，‘stop’）

其中，Wn 是一个二维矢量，Wn=[W1 W2]，W1 和 W2 分别对应于阻带的低端频率和高端频率。

（5）采用其他窗函数设计低通 FIR 滤波器：

不指定窗函数时，FIR1 采用默认的 Hamming 窗。此外，还可以采用其他窗函数，如 Boxcar，Hanning，Bartlett，Blackman，Kasier 和 Chebwin。例如，下面就是一个采用 Bartlett 窗设计 FIR 低通滤波器的例子。

B=FIR1（N, Wn，Bartlett（N+1））

MATLAB 提供了非常方便的帮助功能，因此，各种函数的使用方法只要输入 HELP 命令和函数名就可以得到该函数的使用说明。例如输入下列命令就可以得到 FIR1 函数的使用说明：

HELP　FIR1

此外，FREQZ（）函数可以求出传递函数的幅频响应和相频响应，再用 PLOT（）等函数就可以方便地画出滤波器的幅频响应和相频响应曲线。



13.2.2　FIR 滤波器的定点 DSP 实现

13.2.2.1　TMS320C2xx/C5x 实现

TMS320C2xx/C5x 定点 DSP 芯片所提供的单周期乘/累加带数据移动指令和较大的片内 RAM 空间使数字滤波器每个滤波样值的计算可以在一个周期内完成。TMS320C203 内部具有 544 字的 RAM，分为 B0、B1 和 B2 三块。其中 B0 块（256 字）可以用软件编程为数据区（CNFD）或程序区（CNFP），执行 CNFP 后，B0 块映射到程序区的 FF00H ~ FFFFH。采用高效的 MACD 指令，必须用片内 RAM，其中 B0 块必须配置为程序区。

采用 MACD 指令结合 RPTK 指令就可以实现单周期的滤波样值计算：

```
RPTK     N-1
MACD     （程序地址），（数据地址）
```

其中，指令“RPTK N-1”将立即数 *N*-1 装入到重复计数器，使下一条指令重复执行 *N* 次。MACD 指令实现下列功能：

（1）将程序存储器地址装入到程序计数器；

（2）将存于数据区（B1 块）的数据乘以程序区（B0 块）的数据；

（3）将上次的乘积加到累加器；

（4）移动数据，将 B1 块中的数据向高地址移动一个地址；

（5）每次累加后，程序计数器加 1，指向下一个单位脉冲响应样值。

为了使用 MACD 指令，输入样值 *x*（*n*）和滤波器系数 *h*（*n*）必须合理地进行存放。图 13-2 是输入样值 *x*（*n*）和滤波器系数 *h*（*n*）在 TMS320C2xx 内存中的一种存放方法。例 13.1 是高效实现 FIR 滤波方程的 TMS320C2xx 汇编程序。

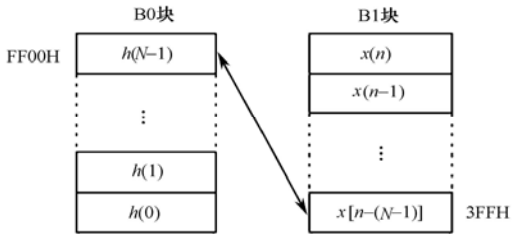


图 13-2　TMS320C2xx 的存储器组织方法

【例 13.1】TMS320C2xx 实现的 FIR 滤波程序

```
;N 阶 FIR 滤波的 TMS320C2xx 程序
;y（n）=x（n−（N−1））h（N−1）+x（n−（N−2））h（N−2）+…+x（n）h（0）
;
        CNFP                    ; B0 块配置为程序区
NEXT    IN      XN,PA0          ; 从 PA0 口取一个样值
        LARP    AR1
        LRLK    AR1,3FFH        ; AR1 指向 B1 块的底部
        MPYK    0               ; P 寄存器清零
        ZAC                     ; ACC 清零
        RPTK    N−1             ; 重复 N−1 次
        MACD    FF00H,*−        ; 乘/累加
        APAC
        SACH    YN,1
        OUT     YN,PA1          ; 输出滤波器响应 y（n）至 PA1 口
        B       NEXT            ; 做下一点滤波
```

程序说明：

（1）FIR 滤波器系数 h（0）,…,h（N-1）均小于 1，可用 Q15 表示。

（2）输入待滤波样值从 PA0 口得到，直接送到 x（n）对应的存储单元。

（3）滤波后的样值由 SACH YN,1 指令送至 YN 存储单元。由于滤波器系数用 Q15 表示，因此乘累加后在 ACC 中的值也是 Q15，左移一位并取高 16 位后得到的数就与输入的样值具有相同的 Q 值了。

（4）滤波后的样值在 PA1 口输出。

（5）一点滤波结束后，由于 MACD 指令的作用，所有的输入样值均向高地址移动一个位置，即 x[n-（N-2）]移动到 x[n-（N-1）]的位置，x[n-（N-3）]移动到 x[n-（N-2）]位置，……，x（n）移动到 x（n-1）的位置，从而为下一点滤波作好准备。

在例 13.1 中，执行 MACD 时用 AR1 间接寻址数据值。采用 RPTK 结合 MACD 实现 FIR 滤波比较适用于滤波器阶数大于 3 的情况。对于阶数小于 3 的 FIR 滤波器，用 LTD/MPY 指令对代替 RPTK/MACD 可以实现更高效率的滤波。

在实际的 DSP 应用中，输入输出的方法不完全相同，上述程序的输入和输出仅是一种实现方法而已。这个程序可用模拟器进行调试。

13.2.2.2　TMS320C54x 实现

TMS320C54x 定点 DSP 提供了单周期乘/累加指令 MAC 和循环寻址方式，使 FIR 数字滤

波器每个样值的计算可以在一个周期内完成。

FIR 数字滤波器每个样值的计算就是实现两数组对应项乘积的累加和。采用 RPTZ 和 MAC 指令结合循环寻址方式可以方便地实现这一运算：

```
RPTZ      累加器，N−1
MAC       （双访问数据），（双访问数据），累加器
```

其中，RPTZ 指令将累加器清零初始化，并将立即数 $N-1$（16 位）装入到重复计数器，使下一条指令重复执行 N 次。MAC 指令实现将两存储区数据的乘积累加到累加器，再通过使存储区指针以循环寻址的方式指向下一个存储区。

为了实现对应项相乘，输入样值 x（n）和滤波器系数 h（n）必须合理存放，并正确初始化这两个存储块和块指针。图 13-3 是输入样值 x（n）和滤波器系数 h（n）在 TMS320C54x 的一种存放方法，图 13-3（a）是滤波器系数的存储器组织，图 13-3（b）是滤波前输入样值的存放顺序，图 13-3（c）是滤波后输入样值的存放顺序。例 13.2 是高效实现 FIR 滤波器的 TMS320C54x 汇编程序。

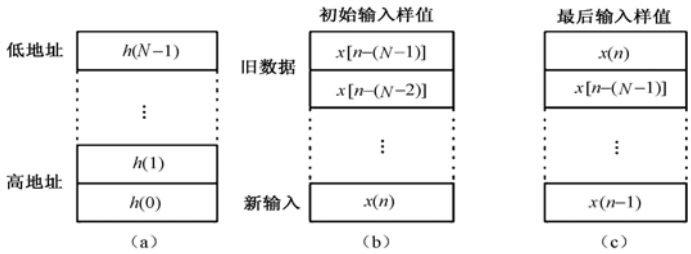


图 13-3　TMS320C54x 实现 FIR 滤波器的存储器组织

【例 13.2】TMS320C54x 实现 FIR 的程序

```
;N 阶 FIR 滤波的 TMS320C54x 程序
;y（n）=x（n-（N-1））h（N-1）+x（n-（N-2））h（N-2）+…+x（n）h（0）
;
    STM     #K_FRAME_SIZE-1,BRC          ; 对每个数据进行滤波
    RPTBD   fir_filter_loop-1
    STM     #K_FIR_BFFR,BK               ; FIR 循环缓冲区大小
    LD      *INBUF_P+,A                  ; 装载输入数据
fir_filter:
    STL     A,*FIR_DATA_P+%              ; 用最新值更新

    RPTZ    A,（K_FIR_BFFR-1）
    MAC     *FIR_DATA_P+%,*FIR_COFF_P+%,A ; 滤波
    STH     A,*OUTBUF_P+
fir_filter_loop
```

程序说明：

（1）数据定标。输入数据和滤波器系数均小于 1，以 $Q15$ 表示。将 FRCT 标志置 1，输入数据和滤波器系数乘完后结果自动左移一位，和累加器相加并取高 16 位输出。这样，输出和输入的 Q 值就一致了。

（2）数据存放的要求。因为采用了 MAC 指令和循环寻址，所以输入数据和滤波器系数的存放要依照一定的要求。数据块和系数块都要放在双寻址数据存储区。并且，起始地址为 m 位地址边界（$2^m>N$）。

（3）循环寻址的使用。为了使用循环寻址，除了对数据的存放有要求外，先还要设置 Bk 为块长 N。由于使用了循环寻址，数据和系数指针在操作后以循环的方式增 1。

（4）数据的初始化。输入数据块要初始化为全 0。

（5）符号说明。常量 K_FRAME_SIZE 和 K_FIR_BFFR 分别为输入数据的个数、FIR 滤波器抽头个数。指针 INBUF_P, FIR_DATA_P, FIR_COFF_P 和 OUTBUF_P 分别为输入数据指针、FIR 抽头输入数据指针、FIR 抽头系数指针、输出数据指针，它们可以用辅助寄存器 ARx 实现。

利用 TMS320C54x 的 FIRS 指令和循环寻址，可以用更简洁的方法实现对称抽头的 FIR 滤波器。这种滤波器具有线性相位的特点，因此得到了广泛的应用。有兴趣的读者可以查看 TI 公司的有关参考资料。

13.2.3　FIR 滤波器的浮点 DSP 实现

TMS320C3x 具有并行乘/累加操作和循环寻址功能，并行乘/累加操作使一次乘法和一次加法在一个指令周期内完成，而循环寻址使长度为 N 的缓冲区足以存储 N 阶滤波所需的输入样值。图 13-4 是输入样值 x（n）和滤波器系数 h（n）在 TMS320C3x 的一种存放方法。其中图 13-4（a）是滤波器系数的存储器组织，图 13-4（b）是滤波前输入样值的存放顺序，图 13-4（c）是滤波后输入样值的存放顺序。

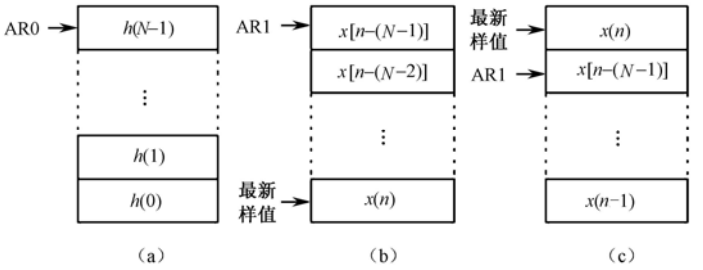


图 13-4　TMS320C3x 实现 FIR 滤波器的存储器组织

为了建立循环寻址，BK 寄存器初始化为滤波器的长度 N，而且输入样值 X 的起始地址必须满足如下条件：起始地址必须是数 M 的整数倍，而 M 是一个大于 N 且是最小的 2 的整数次方的数。例如，若 N=24，则起始地址必须是 32（2 的 5 次方）的整数倍，即起始地址的低 5 位必须为 0。例 13.2 是采用图 13-4 的存储器组织实现 FIR 滤波的 TMS320C3x 汇编子程序。而例 13.3 则是一个完整的 FIR 滤波程序，可用模拟器进行调试。

【例 13.3】TMS320C3x 实现 FIR 滤波器的汇编子程序

```
        .global   FIR
FIR     LDF       0.0,R0                          ; R0 初始化
        LDF       0.0,R2                          ; R2 初始化
        RPTS      N−1                             ; 建立重复
        MPYF3     *AR0++（1）,*AR1++（1）%,R0      ; R0=h（N−1−i）*x[n−（N−1−i）]
        ||ADDF3   R0,R2,R2                        ; 乘/累加
```

```
        ADDF3   R0,R2,R0                    ; 加最后一个乘积
        RETS                                ; 返回
        .end
```

程序说明：

（1）滤波器系数和输入样值的存放顺序如图 13-4 （a）和图 13-4（b）所示。

（2）第一次调用上述程序时，首先要初始化 AR0, AR1, BK 三个寄存器。其中，AR0 指向 $h(N-1)$，AR1 指向 $x[n-(N-1)]$，BK=N，N 为滤波器的阶数。

（3）第二次及以后调用上述程序进行滤波时，AR0 仍需进行赋值，指向 $h(N-1)$。而 AR1 采用循环寻址，不重新初始化。

（4）滤波的结果存放在寄存器 R0 中。

【例 13.4】TMS320C3x 实现 FIR 滤波器的汇编程序

```
STACK_SIZE  .set  40h           ; 定义系统堆栈的大小
N           .set  19            ; 19 点 FIR 滤波
            .data
stack       .usect  ".stack",STACK_SIZE
xn          .usect  "buffer",N
xin_addr    .word   804000h     ; 输入口地址
out_addr    .word   804001h     ; 输出口地址
xn_addr     .word   xn+N-1      ; 输入样值缓冲区最后一个地址
hn_addr     .word   hn          ; h（N−1）的地址
stack_addr  .word   stack       ; 堆栈地址
hn  .float  0.01218354,-0.009012882,-0.02881839,-0.04743239,-0.04584568
    .float  -0.008692503,0.06446265,0.1544655,0.2289794,0.257883
    .float  0.2289794,0.1544655,0.06446265,-0.008692503,-0.04584568
    .float  -0.04743239,-0.02881839,-0.009012882,0.01218354

            .text
begin       LDP     stack_addr
            LDI     @stack_addr,SP
;初始化指针及数组
        LDI     19,BK               ; 设置循环缓冲器
        LDI     @xn_addr,AR1        ; AR1 指向最后一个存储单元
        LDI     @xin_addr,AR2       ; AR2 指向输入口
        LDI     @out_addr,AR3       ; AR3 指向输出口
LOOP    LDF     *AR2,R6             ; 输入一个新的样值
        STF     R6,*AR1++（1）%     ; 存储该样值
        LDI     @hn_addr,AR0        ; AR0 指向 h（N-1）
        CALL    FIR
        STF     R0,*AR3             ; 输出滤波样值
        BR      LOOP
    .end
```

程序说明：

（1）本程序是一个 19 点 FIR 800Hz 低通滤波器。

（2）程序在 804000H 口输入待滤波样值，而在 804001H 口输出滤波后的样值。

（3）初始化时 AR1 指向输入样值缓冲区的最后一个存储单元，最新的输入样值 x（n）存放在这个单元，存储完后，AR1 指向第一个存储单元。第一次滤波结束后，AR1 仍指向第一个单元，因此最新样值就存储在第一个单元，存储完毕，AR1 指向第二个单元，如图 13-4（c）所示。

（4）每次滤波前，AR0 需重新进行初始化，指向 h（N-1）。

13.3　IIR 滤波器的 DSP 实现

13.3.1　IIR 滤波器的基本原理和设计方法

13.3.1.1　IIR 滤波器的特点

IIR 滤波器差分方程的一般形式为：

$$y(n)=\sum_{k=0}^{N-1}a_k x(n-k)+\sum_{k=0}^{N-1}b_k y(n-k) \tag{13-13}$$

IIR 滤波器具有无限长的单位脉冲响应，在结构上存在反馈回路，即是递归型的。也就是说，IIR 滤波器的输出不仅与输入有关，而且与过去的输出有关。滤波器输出 y（n）不仅依赖于当前输入 x（n）及过去输入 x（n-1）,x（n-2）,…,x（n-N+1），而且依赖于过去的输出 y（n-1）,y（n-2）, …,y（n-N+1）。

IIR 滤波器由于具有结构简单、运算量小的特点，因而得到了较广泛的应用。IIR 滤波器结构具有多种形式，归纳起来主要有以下几种。（1）直接 I 型（也称直接型）；（2）直接 II 型（也称正准型）；（3）级联型；（4）并联型；（5）格型（lattice）。直接 I 型结构需要 2N 级延迟单元，直接 II型结构与直接 I 型结构相比节省了一半延迟，即需要 N 级延迟单元，是最常用的 IIR 滤波器结构之一。图 13-5 是直接 II 型 IIR 滤波器的结构。

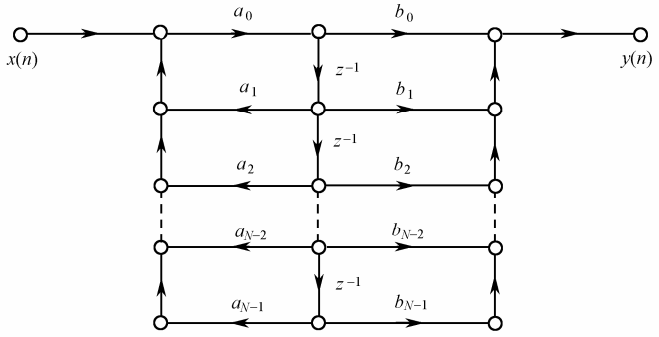


图 13-5　直接II型 IIR 结构

级联型结构采用多个二阶基本节级联而成，每一个二阶基本节可采用直接 II 型结构实现，这种结构便于准确实现滤波器的零极点，也便于性能调整。其转移函数可以表示为

$$H(z)=CH_1(z)H_2(z)\cdots H_r(z) \tag{13-14}$$

并联型结构的转移函数可以表示为

$$H(z)=C+H_1(z)+H_2(z)+\cdots+H_r(z) \tag{13-15}$$

与级联型结构一样，并联型结构中的每个 $H_i(z)$ 也可以全部表示为直接 II 型的结构。以上几种结构是 IIR 滤波器的基本结构。在自适应滤波和语音处理中还有一种应用较广的 IIR 结构是格型结构。格型结构又可分为两种，一种是所谓的全极点结构，另一种是既有极点又有零点的极零结构。全极点 IIR 格型滤波器由多个基本节级联而成，每个基本节如图 13-6 所示。

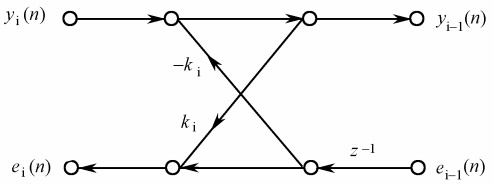


图 13-6　全极点 IIR 格型滤波器的基本节

这种滤波器结构具有一个非常显著的特点，即只要 $|k_i|<1$，i=1,2,…,N，则滤波器可保证稳定。因此，判断全极点格型 IIR 滤波器稳定性的方法就是判断是否满足 $|k_i|<1$ 的条件。

13.3.1.2　IIR 数字滤波器的设计

IIR 滤波器的设计一般有以下两种方法：（1）先设计一个模拟滤波器，然后变换成满足预定指标的数字滤波器。由于模拟滤波器的设计方法很成熟，有许多简单和现成的设计公式，设计参数已经表格化，因而这种方法比较方便。（2）计算机辅助设计法。这是一种最优化的设计方法，先确定一种最佳准则，然后求在此准则下滤波器系统函数的系数。这里我们简单地介绍一下第一种设计方法。

利用模拟滤波器设计理论来设计数字滤波器是一种常用的设计方法。这种方法根据模拟理论设计出满足要求的传递函数 $H_a(s)$，然后再根据 $H_a(s)$ 求得相应的数字滤波器的传递函数 H（z），使得 H（z）的频率响应逼近 $H_a(s)$，同时 H（z）也必须保持 $H_a(s)$ 的因果性和稳定性。

这种设计方法实际上是一个由 s 平面到 z 平面的变换。实现 s 平面到 z 平面变换的常用方法有冲击响应不变方法和双线性变换法两种。由于冲击响应不变法有可能产生频率混叠现象，因此在实际应用中用得最多的是双线性变换法。采用双线性变换法实现 s 平面到 z 平面的变换方法为：

$$s=c\frac{z-1}{z+1} \tag{13-16}$$

式中，c 为常数。设 Ω 代表模拟角频率，$\Omega=2\pi f$，f 是模拟频率，ω 代表数字角频率。将 $s=\mathrm{j}\Omega$ 和 $z=\mathrm{e}^{\mathrm{j}\omega}$ 代入式（13-16），可得

$$\Omega=\tan\left(\frac{\omega}{2}\right) \tag{13-17}$$

由上式可见，经过非线性变换后，Ω 和 ω 成非线性关系，因而相位和频率不再保持直线关系，这是双线性变换法的缺点。虽然双线性变换法存在这样的缺点，但是，由于低通、高通、带通、带阻模拟滤波器都是分段常数型滤波器，只要截止频率映射正确就可以在一定程度上补偿变换中所带来的非线性畸变。

由双线性变换法设计数字滤波器，首先要设计符合指标要求的模拟滤波器，而设计模拟滤波器，一般可查阅现成的数据表格。但模拟滤波器有低通、高通、带通、带阻等各种类型，即使是一种类型，截止频率又各不相同。因此一般只给出归一化原型滤波器的设计数据。归一化原型滤

波器是指截止频率已经归一化成 1 的低通滤波器，而对于截止频率为 Ω_c 的滤波器，可通过适当的变换得到实际的模拟滤波器。例如，对于低通滤波器，用 $\frac{s}{\Omega_c}$ 代替原型滤波器中的 s 即可。

实际应用中，需要设计各种类型的数字滤波器，一般有下列两种方法：（1）把一个归一化模拟滤波器经频带变换为所需类型的模拟滤波器，再通过双线性变换法数字化为所需类型的数字滤波器。实际上，这两步可以合并为一步，即直接由归一化模拟低通滤波器变换到所需类型的数字滤波器。（2）用模拟低通原型线用双线性变换法数字化为数字低通滤波器，再变换成各种类型的数字滤波器。下面我们给出用第一种方法设计各型数字滤波器的一般步骤：

（1）根据数字频率指标计算预畸的模拟频率指标，计算方法为：

$$\Omega = \tan(\frac{\omega}{2})$$

（2）根据通带或阻带允许的波动，确定滤波器类型。根据阻带的衰耗要求，确定滤波器的阶数，进而求出归一化模拟低通的传递函数 $H_a(s)$。

（3）经频率变换得到实际的模拟滤波器传递函数 H（s）。

（4）根据双线性变换法把模拟滤波器 H（s）变换为数字滤波器 H（z）。

上述第 3 步和第 4 步可以合并为一步，直接从归一化模拟低通到数字滤波器的变换方法如表 13-1 所示。

表 13-1　模拟低通原型到各种数字滤波器的变换方法

变换类型	变换方法	说　明
模拟低通原型→ 数字低通	$s=\frac{1}{\Omega_c}\frac{z-1}{z+1}$	$\Omega_c=\tan(\frac{\Omega_c T}{2})$ 表示低通预畸截频
模拟低通原型→ 数字高通	$s=\frac{\Omega_c(z+1)}{z-1}$	$\Omega_c=\tan(\frac{\Omega_c T}{2})$ 表示高通预畸截频
模拟低通原型→ 数字带通	$s=\frac{1}{\Omega_a}\frac{z^2-2\alpha z+1}{z^2-1}$	$\Omega_a=\tan\frac{(\Omega_u-\Omega_l)T}{2}$
模拟低通原型→ 数字带阻	$s=\frac{\Omega_a(z^2-1)}{z^2-2\alpha z+1}$	$\alpha=\frac{\cos[(\Omega_u+\Omega_l)T/2]}{\cos[(\Omega_u-\Omega_l)T/2]}$

注：Ω_u 表示通带或阻带的上截频，Ω_l 表示通带或阻带的下截频，T 为采样周期。

13.3.1.3　设计工具

IIR 滤波器的设计工具与 FIR 滤波器类似，除了可以利用一些专用的滤波器设计工具程序以外，也可以利用 MATLAB 软件来设计 IIR 滤波器。下面我们以设计一个 Chebshev-2 型 IIR 低通滤波器为例来说明用 MATLAB 设计 IIR 滤波器的方法。

设滤波器的通带为 0 ~ Wp，阻带为 Ws ~ 1.0，1.0 对应于半抽样频率，要求通带内波动小于 Rp dB，阻带内衰减大于 Rs dB，且滤波器的传递函数为：

$$H(e^{j\omega})=\frac{B(z)}{A(z)}=\frac{b(1)+b(2)z^{-1}+\cdots+b(nb+1)z^{-nb}}{1+a(2)z^{-1}+\cdots+a(na+1)z^{-na}}$$

则用 MATLAB 设计该滤波器的过程如下：

（1）调用函数 cheb2ord（），确定滤波器的阶数和截止频率：

[N，Wn]=cheb2ord（Wp,Ws,Rp,Rs）

返回值 N 为满足要求的 Chebshev-2 型数字滤波器的最小阶数，Wn 为截止频率；

（2）调用函数 cheby2（）：

[B，A]=cheby2（N,Rs,Wn）

返回值 B 为一个 N+1 维矢量，是分子多项式系数；A 也是一个 N+1 维矢量，为分母多项式系数。

与 FIR1 类似，设计 Chebshev-2 型 IIR 高通、带通和带阻的方法分别如下。

高通：[B，A]=cheby2（N,Rs,Wn, ‘high’）

带通：[B，A]=cheby2（N,Rs,Wn），Wn = [W1 W2]

带阻：[B，A]=cheby2（N,Rs,Wn, ‘stop’）, Wn = [W1 W2]

除了 cheby2 以外，还有 cheby1、butter 等类型。

13.3.2 IIR 滤波器的定点 DSP 实现

在 IIR 滤波器结构中，直接 II 型结构是最常用的滤波器结构，因为这种结构的二阶形式可作为级联型和并联型结构中的基本节。下面我们以这种结构为例，说明其实现方法。二阶直接 II 型结构 IIR 滤波器如图 13-7 所示。

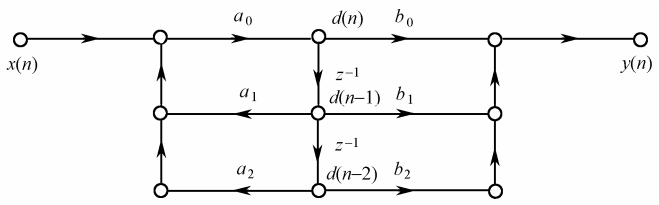


图 13-7　二阶直接 II 型 IIR 滤波器

二阶直接 II 型结构滤波器的差分方程为：

$$d(n) = x(n) + a_1 d(n-1) + a_2 d(n-2)$$

$$y(n) = b_0 d(n) + b_1 d(n-1) + b_2 d(n-2)$$

式中，d（n），d（n-1），d（n-2）对应于图中具有不同延迟的中间节点值。例 13.5 是实现二阶直接 II 型 IIR 滤波器的 TMS320C2xx 汇编程序，例 13.6 是实现二阶直接 II 型 IIR 滤波器的 TMS320C54x 汇编程序。

【例 13.5】TMS320C2xx 实现的 IIR 滤波器汇编程序

```
;本程序实现一个二阶直接 II 型 IIR 滤波器
NEXT    IN      XN,PA0      ;从 PA0 口输入一个样值
        LAC     XN,15       ;ACC=x（n）
        LT      DN1         ;T=d（n−1）
        MPY     A1          ;P=a1*d（n−1）
        LTA     DN2         ;ACC=x（n）+a1*d（n−1）, T=d（n-2）
        MPY     A2          ;P=a2*d（n−2）
        APAC                ;ACC=x（n）+a1*d（n−1）+a2*d（n-2）
        SACH    DN,1        ;d（n）=x（n）+a1*d（n-1）+a2*d（n−2）
        ZAC                 ;ACC=0
        MPY     B2          ;P=b2*d（n-2）
```

```
LTD     DN1          ;T=d（n−1）, ACC = b2∗d（n−2）, d（n−2）=d（n−1）
MPY     B1           ;P=b1∗d（n−1）
LTD     DN           ;T=d（n）,  ACC=b2∗d（n−1）+b1∗d（n-1）, d（n-1）=d（n）
MPY     B0           ;P=b0∗d（n）
APAC                 ;ACC=b2∗d（n−2）+b1∗d（n−1）+b0∗d（n）
SACH    YN,1         ;y（n）=b0∗d（n）+b1∗d（n−1）+b0∗d（n）
OUT     YN,PA1       ;输出滤波样值至 PA1 口
B       NEXT
```

程序说明：

（1）在这个例子中，DN, DN1 和 DN2 分别是二阶基本节中三个中间延迟节点 $d(n)$，$d(n-1)$ 和 $d(n-2)$ 的存储器单元，三个存储单元占据三个连续的存储空间，DN 在低地址，DN2 在高地址，DN1 在中间。这是为了保证连续滤波时 $d(n)$, $d(n-1)$, $d(n-2)$ 三个中间结果正确的存储位置。一点滤波结束后，由于 LTD 指令的作用，$d(n)$ 移动到 $d(n-1)$, $d(n-1)$ 移动到 $d(n-2)$，为下一点滤波作好准备。

（2）A1, A2 和 B0, B1, B2 分别是滤波器系数 a1, a2 和 b0, b1, b2 对应的存储单元。

（3）XN 为新输入样值 $x(n)$ 的存储单元，YN 为滤波后输出样值 $y(n)$ 对应的存储单元。

（4）滤波器的五个滤波器系数均小于 1，可用 Q15 表示。

（5）滤波器的输入样值从 PA0 口输入，LAC XN,15 指令将输入样值 $x(n)$ 左移 15 位后送入累加器 ACC，这是为了与下面乘积相加时保证小数点对齐。

（6）SACH DN,1 和 SACH YN,1 指令将 ACC 中的值左移一位，并将高 16 位存入 $d(n)$ 和 $y(n)$。假设 $x(n)$ 的 Q 值为 0，左移 15 位后送入 ACC，相当于 Q15，与乘积相加后的值也为 Q15。左移一位相当于得到 Q16 的数，而将高 16 位直接存入 $d(n)$ 和 $y(n)$ 又相当于将结果右移 16 位。因此 $d(n)$ 和 $y(n)$ 的 Q 值又变为 0，从而与 $x(n)$ 的 Q 值一致。

【例 13.6】TMS320C54x 实现的 IIR 滤波器汇编程序

```
;本程序实现一个二阶直接 II 型 IIR 滤波器
STM #K_FRAME_SIZE-1,BRC                    ;对每一样值进行滤波
RPTB    iir_filter_loop-1
LD   ∗INBUF_P+,16,A                        ;装入输入数据
iir_filter:
STM     #d_iir_d+2,IIR_DATA_P
STM     #iir_coff_table,IIR_COFF_P
MAC     ∗IIR_COFF_P+,∗IIR_DATA_P-,A    ;input+d（n−2）∗A2
MAC     ∗IIR_COFF_P+,∗IIR_DATA_P-,A    ;input+d（n−2）∗A2+d（n−1）∗A1
STH     A,∗IIR_DATA_P+                 ;d（n）=input+d（n−2）∗A2+d（n−1）∗A1
MAR     ∗IIR_DATA_P+
MPY     ∗IIR_COFF_P+,∗IIR_DATA_P-,A    ;d（n−2）∗B2
MAC     ∗IIR_COFF_P+,∗IIR_DATA_P,A     ;d（n−2）∗B2+d（n−1）∗B1
DELAY   ∗IIR_DATA_P-                   ;d（n−2）=d（n−1）
MAC     ∗IIR_COFF_P+,∗IIR_DATA_P,A     ;d（n−2）∗B2+d（n−1）∗B1+d（n）∗B0
DELAY   ∗IIR_DATA_P-                   ;d（n−1）=d（n）
STH     A, ∗OUTBUF_P+                  ;output=d（n−2）∗B2+d（n−1）∗B1+d（n）∗B0
iir_filter_loop
```

程序说明：

（1）定标说明。滤波器抽头系数的 Q 值可根据实际情况设置，程序中假定为 Q15。

（2）数据存放要求。延迟节点 *d*（n）、*d*（*n*-1）和 *d*（*n*-2）占据三个连续的存储单元，*d*（*n*）在低地址，*d*（*n*-2）在高地址，以 d_iir_d 作首地址。IIR 滤波器系数按 a2, a1, b2, b1, b0 的顺序以地址增的方式存放，以 iir_coff_table 作首地址。

（3）指令说明。MAR 指令实现以辅助寄存器表示的地址指针修改。DELAY 指令通过将操作数表示存储器数据赋给下一个存储单元来更新延迟节点数据。

（4）常量和参数说明。常量 K_FRAME_SIZE 表示输入数据个数。指针 INBUF_P, IIR_DATA_P, IIR_COFF_P, OUTBUF_P 分别表示输入数据指针、IIR 延迟节点指针、IIR 滤波器系数指针和输出数据指针。



13.3.3 IIR 滤波器的浮点 DSP 实现

本节我们以 TMS320C3x 为例说明 IIR 滤波器的浮点汇编语言的高效实现。与 FIR 滤波器的实现一样，并行指令和循环寻址方式可以用于实现 IIR 滤波器以提高运算的速度。这里我们仍以二阶基本节为例说明 IIR 滤波器的 TMS320C3x 的实现方法。下面两个例子中，例 13.7 是一个基本节实现的程序，而例 13.8 则是实现多个二阶基本节级联的程序。

【例 13.7】单个二阶基本节的 TMS320C3x 实现

```
        .text
IIR1    MPYF3   *AR0,*AR1,R0                ; a2*d（n–2） → R0
        MPYF3   *++AR0（1）,*AR1--（1）%,R1 ;b2*d（n–2） → R1

        MPYF3   *++AR0（1）,*AR1,R0         ; a1*d（n–1） → R1
        ||ADDF3 R0,R2,R2                    ;a2*d（n–2）+x（n） → R2

        MPYF3   *++AR0（1）,*AR1--（1）%,R0 ;b1*d（n–1） → R0
        ||ADDF3 R0,R2,R2                    ; a1*d（n–1）+a2*d（n–2）+x（n）→R2

        MPYF3   *++AR0（1）,R2,R2           ;b0*d（n） → R2
        ||STF   R2,*AR1++（1）%             ;存储 d（n），指向 d（n–1）

        ADDF    R0,R2                       ; b1*d（n–1）+b0*d（n） → R2
        ADDF    R1,R2,R0                    ; b2*d（n–2）+b1*d（n–1）+b0*d（n）→ R0
        RETS                                ;返回
        .end
```

程序说明：

（1）滤波器系数 a1、a2 和 b0、b1、b2 以及三个中间延迟节点 d（*n*）、d（*n*-1）和 d（*n*-2）的存放顺序如下所示。为了循环寻址的需要，d（*n*）的起始地址必须是 4 的整数倍，即起始地址的后 2 位必须为 0。

```
AR0—> a2          d（n）
      b2          d（n–1）
      a1      AR1—>d（n–2）
```

b1
b0

（2）程序开始时，AR0 指向系数表包含 a2 的地址，AR1 指向包含延迟节点值 d（n −2）的地址。R2 寄存器包含输入样值 x（n），BK 寄存器初始化为 3。

（3）程序计算过程如下。

- 第一次乘法操作计算

 R0 = a2d（n-2）
- 第二次乘法操作计算

 R1 = b2d（n-2）

 AR0 指向 b2，AR1 指向 d（n-1）；
- 第三条是一条并行指令

 R0=a1d（n−1）

 R2=x（n）+a2d（n-2）

 指令执行后，AR0 指向 a1，AR1 仍然指向 d（n-1）；
- 第四条也是并行指令，

 R0=b1d（n-1）

 R2=x（n）+a2d（n-2）+ a1d（n-1）

 指令执行后，AR0 指向 b1，AR1 指向 d（n）；
- 第五条也是并行指令，

 R2=b0d（n）

 D（n）= x（n）+ a2d（n-2）+ a1d（n-1）

 结束后 AR1 指向 d（n-1）；
- 第六条为加法指令

 R2=b1d（n-1）+ b0d（n）
- 最后一条指令

 R0=b2d（n -2）+ b1d（n -1）+ b0d（n）

 结束后，程序在 R0 中返回滤波结果。

（4）第一点滤波结束后，AR1 指向了 d（n-1），相当于下一点的 d（n-2），为下一点滤波做准备。即：

d（n-1）
AR1→ d（n-2）
d（n）

【例 13.8】N 个二阶基本节级联时的 TMS320C3x 汇编程序

```
;本程序实现 N 个二阶基本节级联时的 IIR 滤波
;方程式为：y（0, n） = x（n）;
;       for （i=0,i<N,i++）
;            {
;             d（i,n）=a2*d（i,n −2）+ a1*d（i,n -1）  *y（i−1,n）;
;             y（i,n）=b2（i）*d（i,n −2） + b1（i）*d（i,n −1）*b0（i）*d（i,n）;
;            }
```

```
;       y（n） =y（N-1,n）;

        .text
IIR2    MPYF3    *AR0,*AR1,R0                ; a2（0）*d（0,n-2） → R0
        MPYF3    *AR1++（1）,*AR1--（1）%,R1  ; b2（0）*d（0,n-2） → R1
        MPYF3    *++AR0（1）,*AR1,R0         ; a1（0）*d（0,n-1） → R0
        ||ADDF   R0,R2,R2                    ; d（0,n）的第一个和项

        MPYF3    *++AR0（1）,*AR1--（1）%,R0 ; b1（0）*d（0,n-1） → R0
        ||ADDF3  R0,R2,R2                    ; d（0,n）的第二个和项
        MPYF3    *++AR0（1）,R2,R2           ; b0（0）*d（0,n） → R2
        ||STF R2,*AR1--（1）%                ; 存储 d（0,n）,指向 d（0,n-2）

        RPTB     LOOP                        ; for 1<=i<n 循环
        MPYF3    *++AR0（1）,*++AR1（IR0）,R0 ; a2（i）*d（i,n-2） → R0
        ||ADDF3  R0,R2,R2                    ; y（i-1,n）的第一个和项
        MPYF3    *++AR0（1）,*AR1--（1）%,R1 ; b2（i）*d（i,n-2） → R1
        ||ADDF3  R1,R2,R2                    ; y（i-1,n）的第二个和项

        MPYF3    *++AR0（1）,*AR1,R0         ; a1（i）*d（i,n-1） → R0
        ||ADDF3  R0,R2,R2                    ; d（i,n）的第一个和项

        MPYF3    *++AR0（1）,*AR1--（1）%,R0 ; b1（i）*d（i,n-1） → R0
        ||ADDF3  R0,R2,R2                    ; d（i,n）的第二个和项

        STF      R2,*AR1--（1）%             ; 存储 d（i,n）,指向 d（i,n-2）
LOOP    MPYF3    *++AR0（1）,R2,R2           ; b0（i）*d（i,n） → R2

        ADDF     R0,R2                       ; y（n-1,n）的第一个和项
        ADDF3    R1,R2,R0                    ; y（n-1,n）的第二个和项
        NOP      *AR1--（IR1）               ; 返回至第一个二阶基本节
        NOP      *AR1--（1）%                ; 指向 d（0,n-1）
        RETS                                 ; 返回
        .end
```

程序说明：

（1）N 个二阶基本节的滤波器系数的存储器组织如下：

滤波器系数	滤波前延迟节点值	滤波后延迟节点值
AR0 →a2（0）	d（0,n）	d（0,n-1）
b2（0）	d（0,n-1）	AR1→ d（0,n-2）
a1（0）	AR1 →d（0,n-2）	d（0,n）
b1（0）	空	空
b0（0）	⋮	⋮
⋮	d（N-1,n）	d（N-1,n-1）
a2（N-1）	d（N-1,n-1）	d（N-1,n-2）
b2（N-1）	d（N-1,n-2）	d（N-1,n）

a1（N-1）	空	空
b1（N-1）		
b0（N-1）		

（2）程序执行前，输入样值在 R2 寄存器中，AR0 指向滤波器系数 a2（0），AR1 指向延迟节点值 d（0,n-2），BK 寄存器初始化为 3，IR0 寄存器初始化为 4，IR1=4N-4，RC=N-2。程序执行结束后，输出结果在 R0 寄存器中。

（3）程序可以分为两个部分，RPTB 循环前的五条指令计算第一个二阶基本节的滤波输出；RPTB 循环体计算剩下的 N-1 个二阶基本节。

（4）程序结束前的两条指令用于调整 AR1 指针，由于滤波结束后，AR1 指向了 d(N-1,n-2），因此执行了指令 NOP　*AR1--（IR1）后，AR1 指向 d（0,n-2），再执行指令 NOP　*AR1--（1）%后，AR1 指向了当前的 d（0,n-1），相当于下一点的 d（0,n-2）。

13.4　自适应滤波器的 DSP 实现

13.4.1　自适应滤波器的基本原理

前面讨论了 FIR 和 IIR 两种具有固定滤波器系数的滤波器，它们的特性都是已知的。但是，在许多 DSP 应用场合，由于无法预先知道信号和噪声的特性或者它们是随时间变化的，因此用这种滤波器显然无法实现最优滤波。在这种情况下，必须设计这样一种滤波器，它具有跟踪信号和噪声变化的能力，使得滤波器的特性也随信号和噪声的变化而变化，以达到最优滤波。这就是具有自学习或训练能力的自适应滤波器。

自适应滤波器的特性变化是由自适应算法通过调整滤波器系数来实现的。一般而言，自适应滤波器由两部分组成，一是滤波器结构，二是调整滤波器系数的自适应算法。图 13-8 是自适应滤波器的一般结构。

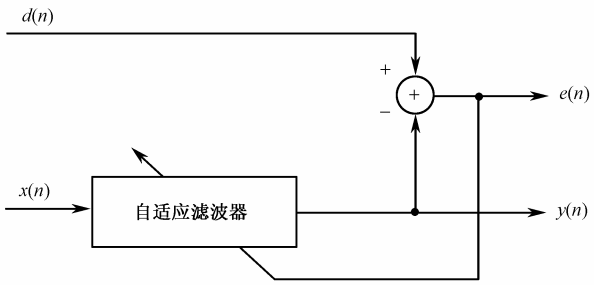


图 13-8　自适应滤波器的一般结构

图 13-8 中，x（n）为输入信号，y（n）为输出信号，d（n）为参考信号或期望信号，e（n）则是 d（n）和 y（n）的误差信号。自适应滤波器的滤波器系数受误差信号 e（n）控制，根据 e（n）的值和自适应算法自动调整。

由于自适应滤波器具有自动调整滤波器系数以适应信号和噪声变化的能力，因此在许多场合得到了广泛的应用。如自适应预测、自适应均衡、自适应回波抵消、自适应噪声抵消等。

自适应滤波器的结构可以采用 FIR 或 IIR 结构，由于 IIR 滤波器存在稳定性问题，因此一

般采用 FIR 滤波器作为自适应滤波器的结构。用 FIR 具体实现时，还可以分为横向结构、对称横向结构和格型结构等几种。

自适应算法的种类也较多，但最常用的算法就是著名的最小均方误差算法，即 LMS 算法。这种算法具有简单、容易实现等特点，应用比较广泛。LMS 算法除了基本的算法之外，有许多变型，如 Sign-error LMS 算法、Sign-data LMS 算法、Sign-sign LMS 算法等。以下就是这几种 LMS 算法的系数迭代公式。

- 基本 LMS：$w(k,n+1) = w(k,n) + u\,e(n)\,x(n-k)$
- Sign-error LMS：$w(k,n+1) = w(k,n) + u\,\mathrm{sgn}[e(n)]x(n-k)$
- Sign-data LMS：$w(k,n+1) = w(k,n) + u\,e(n)\ \mathrm{sgn}[x(n-k)]$
- Sign-sign LMS：$w(k,n+1) = w(k,n) + u\,\mathrm{sgn}[e(n)]\,\mathrm{sgn}[x(n-k)]$

上面几个式子中，$w(k,n)$ 和 $w(k,n+1)$ 分别是迭代前后的系数值，n 和 $n+1$ 表示前后两个时刻，$k=0\sim N-1$，N 为滤波器的阶数。u 为收敛因子，$e(n)$ 是误差信号，$x(n-k)$ 是输入信号。sgn[x]是符号函数，即当 $x\geq 0$ 时，其值为 1，否则为 -1。

由于自适应滤波器要求的实时性强，因此需要高性能的 DSP 芯片来实现。下面以采用横向 FIR 结构和 LMS 算法的自适应滤波器为例，介绍自适应滤波器的 DSP 实现。

13.4.2 自适应滤波器的 TMS320C2xx 定点实现

本小节以 TMS320C2xx 为例讨论自适应滤波器的 DSP 实现。

TMS320C2xx 的指令集中提供了两条功能很强的指令 ZALR 和 MPYA，用这两条指令可以高效地实现系数的迭代。ZALR 指令将存储单元中的数值装入到累加器 ACC 的高 16 位中，并将 ACC 的第 15 位置 1，第 0～14 位置 0。第 15 位置 1 可以减少乘法所带来的截尾噪声。MPYA 指令累加已经在 P 寄存器中的乘积，并且将操作数与 T 寄存器中的数相乘。

设滤波器的系数和输入样值在存储器中的存放顺序如图 13-9 所示，且 ue（n）已经在 T 寄存器中，则实现基本 LMS 算法的 TMS320C2xx 程序为：

```
        LRLK    AR1,N-1          ; 初始化循环计数器
        LRLK    AR2,WN
        LRLK    AR3,XN           ; 指向 x（n-N+1）
        LARP    AR3
        MPY     *-,AR2           ; P=ue（n）x（n-N+1）
ALP0    ZALR    *,AR3            ; 取 w（k,n）
        MPYA    *-,AR2           ; ACC=P+ w（k,n）,P=ue（n）x（n-N+1）
        SACH    *+,0,AR1         ; 存 w（k,n+1）
        BANZ    ALP0,*-,AR2  ;
```

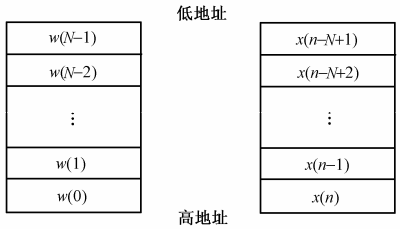


图 13-9　TMS320C2xx 自适应滤波器的存储器组织

下面的例子是 TMS320C2xx 实现 64 点自适应滤波器的汇编程序，自适应算法采用基本的 LMS 算法，滤波器结构采用横向 FIR 结构，收敛因子 u 取 0.01。

【例 13.9】自适应滤波器的 TMS320C2xx 实现

本程序实现 64 阶自适应滤波器，滤波器算法如下：

（1）for（k=0, k<64, k++）　$y(n)=w(k)*x(n-k)$

（2）$e(n)=d(n)-y(n)$

（3）for（k=0, k<64, k++）　$w(k)=w(k)+u*e(n)*x(n-k)$

```
;定义滤波器阶数 ORDER、收敛因子 MU
ORDER:    .equ     64          ; 阶数
MU:       .equ     327         ; Q15,mu=0.01

;定义缓冲器和系数的地址
X0:       .usect   "buffer",ORDER-1
XN:       .usect   "buffer",1
WN:       .usect   "coeffs",ORDER

;定义变量
ONE:      .usect   "vars",1
U:        .usect   "vars",1
ERR:      .usect   "vars”,1
Y:        .usect   "vars",1
D:        .usect   "vars",1
ERRF:     .usect   "vars",1

          .text
START
;初始化
          LDPK     0
          ZAC                        ; ACC=0
          LARP     AR0
          LRLK     AR0,0200h         ; AR0 指向 B0 块
          RPTK     0feh
          SACL     *+                ; 初始化 B0 块
          RPTK     0feh
          SACL     *+                ; 初始化 B1 块
          LACK     1
          SACL     ONE               ; 初始化 ONE = 1
          LALK     MU
          SACL     U                 ; 初始化 U = 327
          SSXM                       ; 置符号扩展
          SPM      1                 ; 置 P 寄存器移位模式
          LARP     AR3
;以下开始进行自适应预测
INPUT
```

```
        IN      D,PA0           ; 从 PA0 口取输入样值
; 估计信号 y（n）
        CNFP                    ; B0 块配置为程序存储器
        MPYK    0               ; P=0
        LAC     ONE,15
        LRLK    AR3,XN          ; 指向 x（n–N+1）
FIR     RPTK    ORDER–1         ; 重复执行 N 次
        MACD    0ff00h,*-       ; 估计 y（n）
        CNFD                    ; B0 块配置为数据块
        APAC
        SACH    Y               ; 存储滤波器的输出
; 插入最新样值
        MAR     *+              ; 指向最新样值的地址
        LAC     D
        SACL    *               ; 插入最新样值
;  计算误差
        SUB     Y               ; ACC = x（n）– y（n）
        SACL    ERR             ; 存储误差值 e（n）
;更新滤波器系数
        LT      ERR             ; T = e（n）
        MPY     U               ; P = u* e（n）
        PAC
        ADD     ONE,15
        SACH    ERRF            ; ERRF = u * e（n）
        LARK    AR1,ORDER-1
        LRLK    AR2,WN          ; AR2 指向滤波器系数
        LRLK    AR3,XN+1        ; AR3 指向数据样值 x（n–N+1），因 MACD
                                ; 已将当前 x（n）移到下一个单元
        LT      ERRF            ; T  = u * e（n）
        MPY     *-,AR2          ; P = u* e（n） * x（n-k）
ALP     ZALR    *,AR3           ; 装入 w（k,n）
        MPYA    *-,AR2          ; w（k,n+1） = w（k,n） + P
                                ; P = u * e（n） * x（n-k）
        SACH    *+,0,AR1        ; 存储 w（k,n+1）
        BANZ    ALP,*-,AR2
 OUT    Y,PA1                   ; 输出信号至 PA1 口
        B       INPUT,*,AR3
      .end
```

对上述程序链接时，需将 coeffs 分配至片内的 B0 块，buffer 分配至片内的 B1 块，而 vars 则分配至 B2 块。

13.4.3 自适应滤波器的 TMS320C54x 定点实现

本小节以 TMS320C54x 为例讨论自适应滤波器的 DSP 实现。

TMS320C54x 提供的 LMS、ST||MPY 和 RPTB 指令为高效地实现自适应滤波提供了方便。

LMS 指令在实现 MAC 指令功能的同时，实现一个带四舍五入的加法。这样，将 LMS、ST||MPY 和 RPTB 合用可以实现在计算当前自适应滤波器输出的同时，更新滤波器的系数。对每一个给定的时刻 n，ue（n）为常数。这个因子保存在临时寄存器 T 中，供一次更新使用。

设自适应滤波器的系数和输入样值在存储器中的存放顺序如图 13-10 所示，且 ue（n）已经存于 T 寄存器中，则实现基本 LMS 算法的 TMS320C54x 程序为：

```
ST          A,*W_COFF_P+
|| MPY      *XW_DATA_P+0%,A          ; A = ue（n）x（n-k）
LMS         *W_COFF_P,*XW_DATA_P     ; A = w（k,n） + ue（n）x（n-k）
                                     ; B = B + w（k）x（n-k）
```

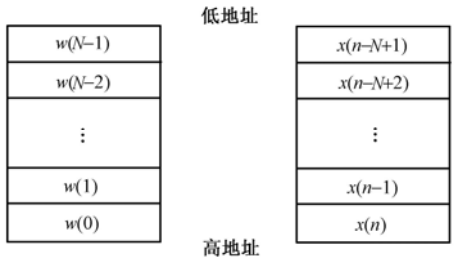


图 13-10　TMS320C54x 自适应滤波器的存储器组织

下面的例子是 TMS320C54x 实现自适应滤波器抵消普通 FIR 滤波器输出的汇编程序，自适应算法采用基本 LMS 算法，滤波器结构采用横向 FIR 结构。自适应滤波器的抽头数以 ADPT_FILT_SIZE 表示，横向滤波器的抽头数以 H_FILT_SIZE 表示。自适应滤波器抽头数可以根据实际需要选择。自适应的对象也可以用不同的模型表示。

【例 13.10】自适应滤波器的 TMS320C54x 实现

```
; 代码分为两个子程序：adapt_init 完成地址指针、存储单元的初始化，adapt_task 完成 LMS 滤波。

;-------------------------------------------
;                   初始化
;-------------------------------------------
.asg        AR0,INDEX_P         ; memory index
.asg        AR1,INIT_P          ; initialize buffer pointer
.asg        AR3,XH_DATA_P       ; data buffer pointer xh（n）
.asg        AR5,XW_DATA_P       ; data buffer pointer xw（n）

.sect       "filter"
adapt_init:
;initialize input data location,input to hybrid,with zero
        STM     #xh,INIT_P
        RPTZ    A,#H_FILT_SIZE-1
        STL     A,*INIT_P+
;initialize input data location,input to adaptive filter,with zero
        STM     #xw,INIT_P
        RPTZ    A,#ADPT_FILT_SIZE-1
        STL     A,*INIT_P+
```

```
;initialize adaptive coefficient with zero
        STM     #wcoff,INIT_P
        RPTZ    A,#ADPT_FILT_SIZE-1
        STL     A,*INIT_P+
;initialize temporary storage locations with zero
        STM     #d_primary,INIT_P
        RPTZ    A,#6
        STL     A,*INIT_P+
;copy system coefficient into RAM location,Reverse order.
        STM     #hcoff,INIT_P          ; it is set in a 8-bit bound address
        RPT     #H_FILT_SIZE-1
        MVPD    #scoff,*INIT_P+

        ST      #K_mu,*（d_mu）
        STM     #1,INDEX_P             ; increment value to be used by dual address

            ;associate auxiliary registers for circular computation
        STM     #xh+H_FILT_SIZE-1,XH_DATA_P
                                               ; point to last input of hybrid buffer
        RETD
        STM     #xw+ADPT_FILT_SIZE-1,XW_DATA_P      ; point to las
;-------------------------------------------
;                   LMS 滤波实现
;-------------------------------------------
.asg    AR2,H_COFF_P            ; H（z）系数缓冲区指针
.asg    AR3,XH_DATA_P           ; 数据系数缓冲区指针
.asg    AR6,INBUF_P             ; 输入缓冲区地址指针
.asg    AR7,OUTBUF_P            ; 输出缓冲区地址指针
.asg    AR4,W_COFF_P            ; W（z） 系数缓冲区指针
.asg    AR5,XW_DATA_P           ; 数据系数缓冲区指针
.sect   "filter"
adapt_task:
        STM     #H_FILT_SIZE,BK     ; first circular buffer size
        STM     #hcoff,H_COFF_P     ; H_COFF_P → last of sys coef
        ADDM    #1,*（d_adapt_count）
        LD      *INBUF_P+,A             ; 装入输入样值
        STM     #wcoff,W_COFF_P         ; reset coeff buffer
        STL     A,*（d_new_x）          ; 读入新数据
        STL     A,*XH_DATA_P+0%         ; 存入缓冲区
        NOP                             ; 避免流水线冲突
        RPTZ    A,#H_FILT_SIZE-1    ; 重复 128 次
        MAC     *H_COFF_P+0%,*XH_DATA_P+0%,A ; 乘累加: a=a+（h*x）
        STH     A,*（d_primary）                    ;primary signal
;开始同时进行滤波和更新自适应滤波器
        LD      *（d_mu_e）,T                   ; T=step_size*error
```

```
SUB     B,B                              ; B 寄存器清 0
STM     #（ADPT_FILT_SIZE-3）,BRC        ; 设置块重复计数器
STM     #ADPT_FILT_SIZE,BK               ; second circular buffer size
RPTBD   lms_end-1
MPY     *XW_DATA_P+0%,A                  ; error*oldest sample
LMS     *W_COFF_P,*XW_DATA_P             ; B=filtered output（y）
;更新滤波器系数
ST      A,*W_COFF_P+                     ; 存储更新后的滤波器系数
|| MPY  *XW_DATA_P+0%,A                  ; error*x[n-（N-1）]
LMS     *W_COFF_P,*XW_DATA_P             ; B=accum filtered output y
lms_end
ST      A,*W_COFF_P+                     ; ASM=0
|| MPY  *XW_DATA_P+0%,A
MVKD    #d_new_x,*XW_DATA_P              ; 存储最新样
LMS     *W_COFF_P,*XW_DATA_P+0%          ; X（0）*h（0）.XW_DATA_P
;point to oldest data
STH     A,*W_COFF_P+                     ; final coeff.W_COFF_P point
                                                 ; to table start
STH     B,*（d_output）                  ; 存储滤波后的输出
LD      *（d_primary）,A
SUB     *（d_output）,A
STL     A,*（d_error）                   ; 存储残留误差信号
LD      *（d_mu）,T
MPY     *（d_error）,A                   ; A=u*e
STH     A,*（d_mu_e）                    ; save the error*step_size
LD      *（d_error）,A                   ; 残留误差信号
STL     A,*OUTBUF_P+
LD      #K_FRAME_SIZE,A                  ; 检查一帧样值是否处理完
SUB     *（d_adapt_count）,A
BC      adapt_task,AGT
ST      #0,*（d_adapt_count）            ; 恢复记数
RET
.end
```

程序说明：

（1）数据存放要求。因为采用了循环寻址，必须将普通 FIR 滤波器和自适应滤波器的抽头系数和抽头输入数据置于 8 位地址边界。

（2）常量说明。H_FILT_SIZE, ADPT_FILT_SIZE, K_FRAME_SIZE, K_mu 分别表示普通 FIR 滤波器抽头数、自适应滤波器抽头数、输入数据个数和 u。

（3）变量和地址说明。变量 d_adapt_count, d_new_x, d_primary, d_output, d_mu, d_error, d_mu_e 分别表示输入数据计数器、新输入的数据、普通滤波器输出、自适应滤波器输出、u、补偿误差、ue（n）。其中，d_adapt_count 用于实现双重循环。地址 hcoff, wcoff, xh, xw 分别表示普通滤波器抽头系数表和自适应滤波器抽头系数表首地址及其抽头输入数据首地址。

13.4.4 自适应滤波器的 TMS320C3x 浮点实现

虽然 TMS320C3x 没有提供专门用于自适应滤波器系数更新的指令，但是采用块重复指令 RPTB 和并行指令仍然可以用两条指令实现系数更新。例 13.11 给出了用 TMS320C3x 汇编语言实现 64 阶 LMS 自适应滤波器的程序，滤波算法与例 13.9 相同，可在模拟器上进行调试。

【例 13.11】自适应滤波器的 TMS320C3x 实现

```
STACK_SIZE      .set    40h                 ; 定义系统堆栈的大小
order           .set    64                  ; 滤波器的阶数
mu              .set    0.01                ; 收敛因子

                .data
stack           .usect  ".stack",STACK_SIZE
xn              .usect  "buffer",order
wn              .usect  "coeffs",order
din_addr        .word   0804000h            ; 输入 d（n）的口地址
xin_addr        .word   0804001h            ; 输入 x（n）的口地址
out_addr        .word   0804002h            ; 输出 y（n）的口地址
xn_addr         .word   xn
wn_addr         .word   wn
u               .float   mu
stack_addr  .word  stack                    ; 堆栈地址

                .text
begin           LDP     stack_addr
                LDI     @stack_addr,SP

;初始化指针及数组
                LDI     order,BK            ; 设置循环缓冲器
                LDP     @xn_addr            ; 设置数据页指针
                LDI     @xn_addr,AR0        ; AR0 指向 x[]
                LDI     @wn_addr,AR1        ; AR1 指向  w[]
                LDF     0.0,R1              ; R1 = 0.0
                RPTS    order-1
                STF     R1,*AR0++（1）%     ; x[] = 0.
                ||STF   R1,*AR1++（1）%     ; w[] = 0.
                LDI     @din_addr,AR5       ; 设置 d（n）输入口地址
                LDI     @xin_addr,AR6       ; 设置 x（n） 输入口地址
                LDI     @out_addr,AR7       ; 设置 y（n）输出口地址
; 以下开始进行自适应预测
input:
                LDF     *AR5,R7             ; 输入 d（n）
                ||LDF   *AR6,R6             ; 输入 x（n）
                STF     R6,*AR0             ; 将 x（n） 加入到缓冲器
; 计算滤波器的输出 y（n）
```

```
                LDF     0.0,R2                  ; R2 = 0.0
                MPYF3   *AR0++（1）%,*AR1++（1）%,R1
                RPTS    order-2
                MPYF3   *AR0++（1）%,*AR1++（1）%,R1
                ||ADDF3 R1,R2,R2
                ADDF    R1,R2                   ; 加最后一个结果
; 计算误差信号 e（n）
                SUBF    R2,R7                   ; e（n） = d（n） －y（n）
; 输出 y（n） 信号
                STF     R2,*AR7                 ; 输出 y（n）
; 更新 w（n）
                MPYF    @u,R7                   ; R7 = err = e（n） * u
                MPYF3   *AR0++（1）%,R7,R1      ; R1 = err * x（n）
                LDI     order-3,RC              ; 初始化重复计数器
                RPTB    WEIGHT                  ; do k = 0, N−3
                MPYF3   *AR0++（1）%,R7,R1      ; R1 = err * x（n−k−1）
                ||ADDF3 *AR1,R1,R2              ; R2 = w（k,n） + err * x（n−k）
WEIGHT  STF     R2,*AR1++（1）%                 ; 更新 w（k,n+1）
                MPYF3   *AR0,R7,R1              ; for k= N − 2
                ||ADDF3 *AR1,R1,R2
                BD      input                   ; 延迟跳转
                STF     R2,*AR1++（1）%         ; 更新 w（N−2,n+1）
                ADDF3   *AR1,R1,R2
                STF     R2,*AR1++（1）%         ; 更新最后一个系数
                .end
```

13.5　小　　结

本章简要介绍了 FIR、IIR 和自适应滤波器的基本原理和设计方法，举例说明了三种滤波器的定点和浮点 DSP 的快速实现方法。这些例子对于实现三种滤波器的其他变型具有很好的借鉴作用。

13.6　习题与思考题

（1）FIR 和 IIR 滤波器有哪些设计方法？每种方法的设计步骤是什么？

（2）设计数字滤波器可以借助 MATLAB 软件，熟悉 MATLAB 软件的使用方法，并用该软件设计一个低通滤波器，参数自定。

（3）用 TMS320C54x 实现数字滤波器有哪些高效指令？试用 TMS320C54x 汇编语言实现上题所述的低通滤波器。

第 14 章　FFT 算法的 DSP 实现

14.1　引　　言

傅里叶变换是一种将信号从时域变换到频域的变换形式，是声学、语音、电信和信号处理等领域中一种重要的分析工具。离散傅里叶变换（DFT）是连续傅里叶变换在离散系统中的表示形式。由于 DFT 的计算量很大，因此在很长一段时间内其应用受到很大的限制。快速傅里叶变换（FFT）是快速计算 DFT 的一种高效方法，FFT 的出现使 DFT 的运算大大简化，运算时间缩短 1~2 个数量级之多，从而使 DFT 在实际应用中得到了广泛的应用。

DSP 芯片的出现使 FFT 的实现变得更为方便。由于多数 DSP 芯片都能在一个指令周期内完成一次乘法和一次加法，而且提供专门的 FFT 指令（如实现 FFT 所必须的比特反转等），使得 FFT 算法在 DSP 芯片上实现的速度更快。本章首先简要介绍 FFT 算法的基本原理，然后分别介绍 FFT 算法在定点 DSP 芯片和浮点 DSP 芯片上的实现方法和程序实例。

14.2　FFT 的基本原理

FFT 是 DFT 的快速计算方法，为了便于理解 FFT 算法，我们先简要地介绍 DFT 的基本原理。

14.2.1　DFT 的基本原理

DFT 是连续傅里叶变换的离散形式。模拟信号 $x(t)$ 的连续时间傅里叶变换（或称频谱）可以表示为：

$$X(\omega)=\int_{-\infty}^{\infty}x(t)\,\mathrm{e}^{-\mathrm{j}\omega t}\mathrm{d}t \tag{14-1}$$

$x(t)$ 经抽样后变为 $x(n\mathrm{T})$，T 为抽样周期。离散信号 $x(n\mathrm{T})$ 的傅里叶变换可以表示为：

$$X(k)=\sum_{n=0}^{N-1}x(n)W_N^{nk},k=0,1,\cdots,N-1 \tag{14-2}$$

式中，$W_N=\mathrm{e}^{-\mathrm{j}2\pi/N}$，称为蝶形因子。式（14-2）实际上就是 N 点的 DFT。由式（14-2）可以看出，计算所有的 $X(k)$ 约需要 N^2 次乘法和 N^2 次加法。可见当 N 较大时，计算 $X(k)$ 的运算量很大。FFT 算法利用了蝶形因子 W_N 内在的对称性和周期性，从而加快了运算的速度。图 14-1 示出了 N=8 时蝶形因子 W_N 的对称性和周期性。

从图中可以看出，

对称性：$W_N^k=-W_N^{k+N/2}$ （14-3）

周期性：$W_N^k=W_N^{N+k}$ （14-4）

FFT 算法将长序列的 DFT 分解为短序列的 DFT。N 点的 DFT 先分解为 2 个 $N/2$ 点的 DFT，

每个 $N/2$ 点的 DFT 又分解为 $N/4$ 点的 DFT，等等。最小变换的点数即所谓的“基数（radix）”。因此，基数为 2 的 FFT 算法的最小变换（或称蝶形）是 2 点 DFT。一般地，对 N 点 FFT，对应于 N 个输入样值，有 N 个频域样值与之对应。

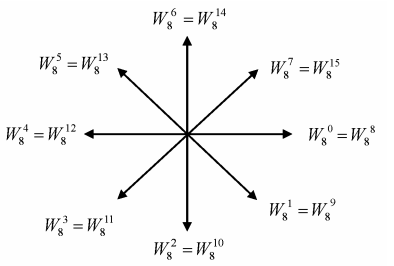


图 14-1　W_N 的对称性和周期性

14.2.2　FFT 算法的导出

一般而言，FFT 算法可以分为时间抽取（DIT）FFT 和频率抽取（DIF）FFT 两大类。下面分别来介绍这两种算法的导出。

1．时间抽取FFT（DIT）

时间抽取 FFT 是将 N 点的输入序列 $x(n)$按照偶数和奇数分解为偶序列和奇序列两个序列：

偶序列：$x(0), x(2), x(4), \cdots, x(N-2)$

奇序列：$x(1), x(3), x(5), \cdots, x(N-1)$

因此，$x(n)$ 的 N 点 FFT 可以表示为：

$$X(k) = \sum_{n=0}^{N/2-1} x(2n) W_N^{2nk} + \sum_{n=0}^{N/2-1} x(2n+1) W_N^{(2n+1)k} \tag{14-5}$$

根据：

$$W_N^2 = [\mathrm{e}^{-\mathrm{j}(2\pi/N)}]^2 = [\mathrm{e}^{-\mathrm{j}2\pi/(N/2)}] = W_{N/2}$$

可得：

$$X(k) = \sum_{n=0}^{N/2-1} x(2n) W_{N/2}^{nk} + W_N^k \sum_{n=0}^{N/2-1} x(2n+1) W_{N/2}^{nk} \tag{14-6}$$

令 $Y(k)$ 和 $Z(k)$ 分别表示式（14-6）中右边第一个和第二个和式，则有

$$X(k) = Y(k) + W_N^k Z(k) \tag{14-7}$$

由于 $Y(k)$和 $Z(k)$的周期为 $N/2$，因此计算上式的 k 的范围为 0～$N/2-1$。计算 $k=N/2$~$N-1$ 的 $X(k)$ 可以利用 $W_N^{k+N/2} = W_N^k$ 的特性，可得

$$X(k + N/2) = Y(k) - W_N^k Z(k) \tag{14-8}$$

式（14-7）和式（14-8）分别用来计算 $0 \leqslant k \leqslant N/2-1$ 和 $N/2 \leqslant k \leqslant N-1$ 的 $X(k)$。以同样的方式进一步抽取，就可以得到 $N/4$ 点的 DFT，重复这个抽取过程，就可以使 N 点的 DFT 用一组 2 点的 DFT 来计算。

在基数为 2 的 FFT 中，设 $N = 2^M$，则总共有 M 级运算，每级中有 $N/2$ 个 2 点 FFT 蝶形运算，因此，N 点 FFT 总共有 $(N/2)\log_2 N$ 个蝶形运算。

基 2 DIT FFT 的蝶形运算如图 14-2 所示。

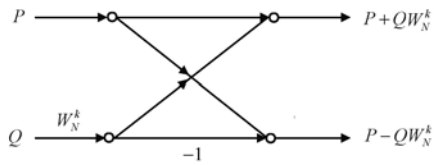


图 14-2　基 2 DIT FFT 蝶形运算

设蝶形的输入分别为 P 和 Q，输出分别为 P' 和 Q'，则有

$$P' = P + QW_N^{k'} \tag{14-9}$$

$$Q = P - QW_N^k \tag{14-10}$$

图 14-3 是 8 点 DIT FFT 的信号流图。

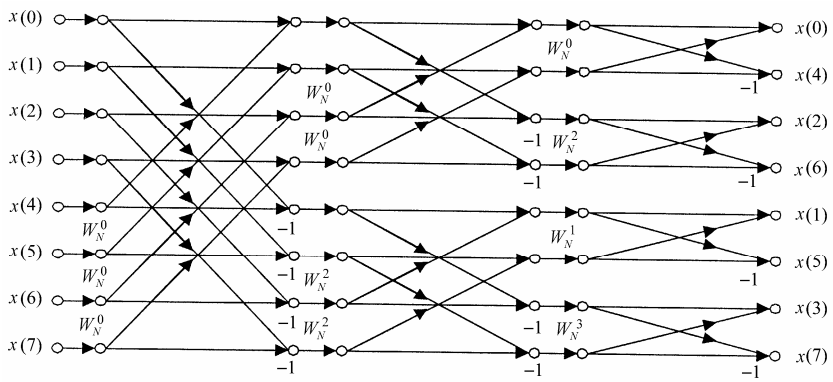


图 14-3　8 点 DIT FFT 的信号流图

从图 14-3 可以看出，输出序列是按自然顺序排列的，而输入序列的排列顺序则是置乱的。但置乱是有一定规律的，这个规律就是所谓的“码位倒置”，或称“比特反转”。也就是说，如果将置乱序列的下标用二进制表示，然后将二进制数按相反方向排列，即可得到这个序列的实际位置。例如，x（3）在输入端的第六个位置（从 0 开始计数），其下标为 $3_{10} = 011_2$，比特反转后得 $110_2 = 6_{10}$，即是 x（3）的实际位置。第三个位置由 x（6）占用，因此，输入端只需将 x（3）和 x（6）交换位置，即可得到正确的位置排列，其余类推。

如果输入序列 x（n）是按自然顺序排列，则输出 X（k）就是按比特反转的规律排列的，如图 14-4 所示。

2．频率抽取FFT（DIF）

时间抽取 FFT 算法的特点是每一级处理都是在时域里把输入序列依次按奇、偶一分为二分成较短的序列。与此类似，还有一种算法是在频域里把序列分解为奇、偶的形式来进行计算，这就是所谓的频率抽取 FFT 算法。

频率抽取 FFT 算法首先将输入序列按照自然顺序分为前半部分和后半部分：

前半部分：x（0）,x（1）,x（2）,…,x（N/2-1）

后半部分：x（N/2）,x（N/2+1）,…,x（N-1）

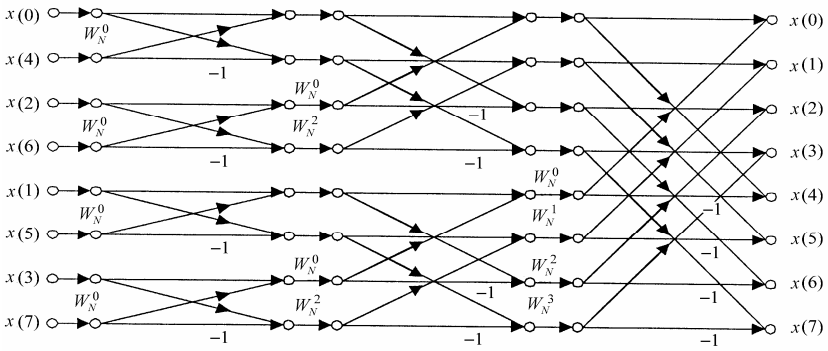


图 14-4　8 点 DIT FFT 的另一种信号流图

则 x（n）的 DFT 可以表示为：

$$\begin{aligned} X(k) &= \sum_{n=0}^{N/2-1} x(n)W_N^{nk} + \sum_{n=N/2}^{N-1} x(n)W_N^{nk} \\ &= \sum_{n=0}^{N/2-1} x(n)W_N^{nk} + W_N^{kN/2}\sum_{n=0}^{N/2-1} x(n+N/2)W_N^{nk} \\ &= \sum_{n=0}^{N/2-1} x(n)W_N^{nk} + (-1)^k \sum_{n=0}^{N/2-1} x(n+N/2)W_N^{nk} \\ &= \sum_{n=0}^{N/2-1} [x(n)+(-1)^k x(n+N/2)]W_N^{nk} \end{aligned} \tag{14-11}$$

由于式（14-11）存在 k 为奇数和偶数两种形式，因此，有可能采取类似 DIT 的方法，在频率域内进行分解。

$$X(2k) = \sum_{n=0}^{N/2-1} [x(n)+x(n+N/2)]W_N^{2nk}, k = 0,1,\cdots,(N/2-1) \tag{14-12}$$

$$X(2k+1) = \sum_{n=0}^{N/2-1} [x(n)-x(n+N/2)]W_N^n W_N^{2nk}, k = 0,1,\cdots,(N/2-1) \tag{14-13}$$

因为 $W_N^{2nk} = W_{N/2}^{nk}$，令 y（n）=x（n）+x（n+N/2），z（n）=x（n）-x（n+N/2），则有：

$$X(2k) = \sum_{n=0}^{N/2-1} y(n)W_{N/2}^{nk} \tag{14-14}$$

$$X(2k+1) = \sum_{n=0}^{N/2-1} z(n)W_N^n W_{N/2}^{nk} \tag{14-15}$$

X（2k）和 X（2k+1）分别为偶数和奇数的 X（k）。以同样的方式，就可以得到 N/4 点的 DFT，重复这个过程，就可使 N 点的 DFT 用一组 2 点的 DFT 来计算。设 $N = 2^M$，则共有 M 级运算。DIF 的基 2 蝶形运算如图 14-5 所示。

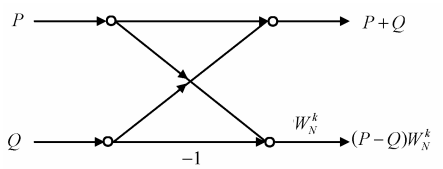


图 14-5　基 2 DIF FFT 蝶形运算

图 14-6 和图 14-7 分别是 8 点 DIF FFT 的两种信号流图方式。

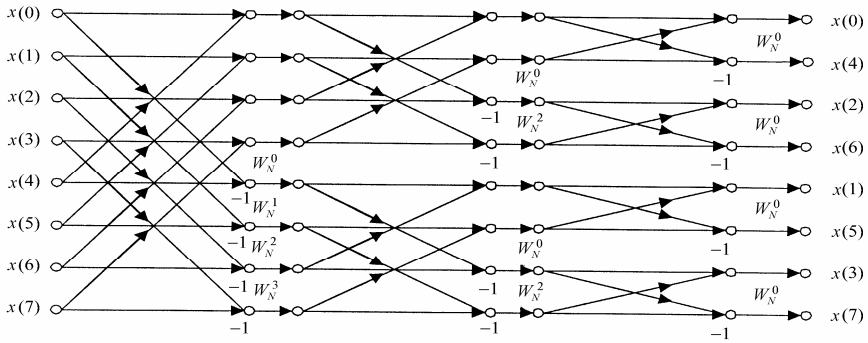


图 14-6　8 点基 2 DIF FFT 的流图

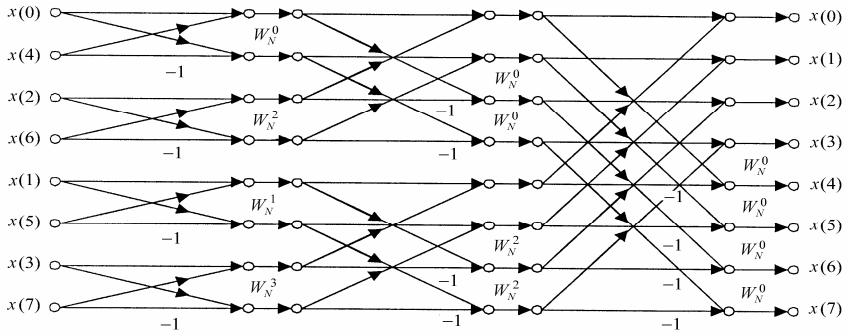


图 14-7　8 点基 2 DIF FFT 的另一种流图

从图 14-2 和图 14-5 可以看出，DIT 和 DIF 两种 FFT 算法的区别是蝶形因子 W_N^k 出现的位置不同，DIF 中 W_N^k 在输入端，而 DIF 中 W_N^k 在输出端，除此之外，两种方法是一样的。具体应用时，可以根据实际情况进行选择。

3．基 4 FFT

上面我们讨论的是常用的基 2 FFT 算法，蝶形的输入和输出均为 2 个。如果采用更高基数的 FFT，则 FFT 的运算速度可以进一步加快。但当基数大于 4 时 FFT 的运算速度提高不多，因此，基 2 和基 4 是两种最常用的方法。

在基 4 FFT 中，蝶形的输入和输出均为 4 个。一个基 4 蝶形包含了 2 级基 2 蝶形（4 个基 2 蝶形）。一般而言，采用基 4 FFT 算法，要比基 2 FFT 算法的速度提高 20%左右。有关基 4 FFT 的公式推导，感兴趣的读者可参阅数字信号处理的有关文献，这里就不作介绍了。

4．用FFT计算反FFT

反 FFT 运算可以表示为：

$$x(n)=\frac{1}{N}\sum_{k=0}^{N-1}X(k)W_N^{-nk},n=0,1,2,\cdots,N-1 \qquad (14\text{-}16)$$

式中，X（k）是时域信号 x（n）的傅里叶变换，比较式（14-16）和式（14-2）可以看出，通过下列修改，我们可以用 FFT 算法来实现反 FFT：

（1）增加一个归一化因子 1/N；

（2）将 W_N^{nk} 用其复共轭 W_N^{-nk} 代替。

由于第二点需要修改符号，因此 FFT 程序还不能不加修改地来计算反 FFT。

因为

$$x(n)=\frac{1}{N}[\sum_{k=0}^{N-1}X^*(k)W_N^{nk}]^* \\ =\frac{1}{N}\{\mathrm{FFT}[X^*(k)]\}^* \tag{14-17}$$

可见，求 X（k）的反 FFT 可以分为以下三个步骤：

（1）取 X（k）的共轭，得 $X^*(k)$；

（2）求 $X^*(k)$ 的 FFT，得 $Nx^*(n)$；

（3）取 $x^*(n)$ 的共轭，并除以 N，即得 x（n）。

采用这种方法可以完全不用修改 FFT 程序就可以计算反 FFT。

5．实数序列的FFT

上面讨论的 FFT 算法都是在输入和输出均为复数序列的条件下得出的。在实际的应用中，输入序列通常是实数序列。计算实数序列的 FFT 当然可以用复数 FFT 算法，只要将虚部置零即可。但是，如果能够考虑是序列 FFT 的对称特性，则实序列 FFT 的运算量可以降低一半，同时存储量也可减少一半。

设实序列 x（n）的 FFT 输出为 X（k），且 X（k）可以表示为

$$X（k）=R（k）+\mathrm{j}I（k） \tag{14-18}$$

且序列长度为 N，则 R（k）和 I（k）满足下列等式：

$$R（k）=R（N-k），k=1，\cdots，N/2-1 \tag{14-19}$$

$$I（k）=-I（N-k），k=1，\cdots，N/2-1 \tag{14-20}$$

$$I（0）=I（N/2）=0 \tag{14-21}$$

也就是说，FFT 变换的实部关于零频率是对称的，而虚部是反对称的。计算实数 FFT 之所以能够节省一半的运算量，是由于只需计算一半的 FFT 值，因此存储量也可以节省一半。

14.3　FFT 算法的 C 语言实现

由于 DSP 程序经常需要用 C 语言来模拟算法的性能，在用到 FFT 算法时当然也需要用 C 语言来实现，例 14.1 是一个用 C 语言实现的时间抽取 FFT 算法程序。程序的输入有两个参数，一个是输入数据序列 xin，采用复数结构，数组从下标为 1 开始存放；另一个是 FFT 的点数 N。FFT 的输出也存放在从下标为 1 开始的复数结构 xin 中。设输入复数序列在结构 s 中，则调用 FFT 子程序求 s 的 256 点 FFT 的方法为：

FFT（s,256）；

【例 14.1】FFT 算法的 C 语言实现

```
#include   <math.h>
struct   compx   { float real,imag;};                    /*定义一个复数结构*/
struct   compx   s[257];                                 /*FFT 输入和输出：均从 s[1]开始存放*/
struct   compx EE（struct compx ,struct compx ）;  /*定义复数相乘结构*/
```

```
void      FFT（struct compx *, int ）                    /*FFT 函数定义*/

struct   compx EE（struct compx b1,struct compx b2）
{struct compx b3;
 b3.real=b1.real*b2.real-b1.imag*b2.imag;
 b3.imag=b1.real*b2.imag+b1.imag*b2.real;
 return（b3）;
}

/*输入 ：xin（实部，虚部），输出：xin（实部，虚部） ，N：FFT 点数*/
void      FFT（struct compx *xin, int N）
{
 int f,m,nv2,nm1,i,k,j=1,l;
 struct compx v,w,t;
 nv2=N/2;
 f=N;
 for （m=1; （f=f/2）!=1;m++） {;}
 nm1=N−1;
 /*变址运算*/
  for （i=1;i<=nm1;i++）
   { if（i<j） { t=xin[j];xin[j]=xin[i];xin[i]=t;}
     k=nv2;
     while（k<j）{j=j-k;k=k/2;}
     j=j+k;
   }
 /*fft*/
 {int le,lei,ip;
  float pi,x,y;
  for （l=1;l<=m;l++）
      {le=pow（2,l）;
       lei=le/2;
       pi=3.14159265;
       v.real=1.0;v.imag=0.0;
       w.real = cos（pi/lei）; w.imag= -sin（pi/lei）;
       for （j=1;j<=lei;j++）
           { for （i=j;i<=N;i=i+le）
               {ip=i+lei;
                t=EE（xin[ip],v）;
                xin[ip].real=xin[i].real-t.real;
                xin[ip].imag=xin[i].imag-t.imag;
                xin[i].real=xin[i].real+t.real;
                xin[i].imag=xin[i].imag+t.imag;
               }
             v=EE（v,w）;
           }
       }
  }
}
```

14.4　FFT 的定点 DSP 实现

14.4.1　运算溢出及避免方法

用定点 DSP 芯片实现 FFT 程序时，一个比较重要的问题是需要防止中间结果的溢出。防止中间结果溢出的方法是对中间数值归一化。下面我们来看一看溢出是如何产生的。

根据 N 点 DFT 的公式，应用 Parseval 定理可得：

$$\sum_{k=0}^{N1} x^2(n) = \frac{1}{N}\sum_{k=0}^{N-1}\left|X(k)\right|^2 \tag{14-22}$$

或

$$N[\frac{1}{N}\sum_{k=0}^{N-1} x^2(n)] = [\frac{1}{N}\sum_{k=0}^{N-1}\left|X(k)\right|^2] \tag{14-23}$$

也就是说，X（k）的均方值是输入 x（n）均方值的 N 倍。因此，计算 x（n）的 DFT 时，如果没有合适的归一化，溢出是不可避免的。下面我们进一步来看看在 FFT 计算过程中溢出是如何产生的。考虑 N 点 FFT 第 m 级的基 2 蝶形（参考图 14-2），输出可以表示为：

$$\begin{aligned} \mathrm{Pm}+1 &= \mathrm{Pm} + W_N^k \mathrm{Qm} \\ \mathrm{Qm}+1 &= \mathrm{Pm} - W_N^k \mathrm{Qm} \end{aligned} \tag{14-24}$$

这里，Pm 和 Qm 是输入，Pm+1 和 Qm+1 是输出。一般地，Pm,Qm,Pm+1 和 Qm+1 与蝶形因子一样是复数。蝶形因子可以表示为：

$$W_N^k = \mathrm{e}^{-\mathrm{j}(2\pi/N)k} = \cos(X) - \mathrm{j}\sin(X) \tag{14-25}$$

式中，X=（2π/N）k。这样 Pm 和 Qm 可以用实部和虚部表示为：

$$\begin{aligned} &\mathrm{Pm}=\mathrm{PR}+\mathrm{jPI} \\ &\mathrm{Qm}=\mathrm{QR}+\mathrm{jQI} \end{aligned} \tag{14-26}$$

将式（14-25）和式（14-26）代入式（14-24），可得：

$$\begin{aligned} \mathrm{Pm}+1 &= \mathrm{PR}+\mathrm{jPI}+（\mathrm{QRcos}（X）+\mathrm{QIsin}（X））+\mathrm{j}（\mathrm{QIcos}（X）-\mathrm{QRsin}（X）） \\ &=（\mathrm{PR}+\mathrm{QRcos}（X）+\mathrm{QIsin}（X））+\mathrm{j}（\mathrm{PI}+\mathrm{QIcos}（X）-\mathrm{QRsin}（X）） \\ \mathrm{Qm}+1 &= \mathrm{PR}+\mathrm{jPI}-（\mathrm{QRcos}（X）+\mathrm{QIsin}（X））-\mathrm{j}（\mathrm{QIcos}（X）-\mathrm{QRsin}（X）） \\ &=（\mathrm{PR}-\mathrm{QRcos}（X）-\mathrm{QIsin}（X））+\mathrm{j}（\mathrm{PI}-\mathrm{QIcos}（X）+\mathrm{QRsin}（X）） \end{aligned} \tag{14-27}$$

假设每个蝶形的输入用 Q15 表示，幅度小于 1，则式（14-27）输出的最大幅度为

$$1＋1\sin（45°）+1\cos（45°）=2.414\ 213\ 562$$

为了避免溢出，可在 FFT 的每一级用因子 2.414 213 562 进行归一化。但是，每一级用这样一个因子归一化势必增加运算量。考虑到大多数情况是实数 FFT，式（14-27）的最大幅度不超过 2，因此可在每一级用因子 2 进行归一化。运用 DSP 芯片的移位特性，用 2 归一化不增加任何运算量。这样，如果 FFT 包含 M 级，则输出相当于除以 $2^M = N$，N 为 FFT 的长度。

值得指出的是，为了避免溢出而对每一级都进行归一化会降低运算的精度。因此，最好的方法是只有对可能溢出的进行归一化，而不可能溢出的则不进行归一化。

14.4.2 FFT 的 TMS320C2xx 实现

FFT 的运算时间往往被用来衡量 DSP 芯片性能的一个重要指标，因此，提高 FFT 的运算速度也是非常重要的。在用 DSP 芯片实现 FFT 算法时，应充分利用 DSP 芯片所提供的各种软硬件资源。对于 TMS320C2xx，可供利用的一些资源包括：①片内 RAM。片内 B0 和 B1 块可以实现 256 点复数 FFT；②比特反转寻址方式。这种寻址方式是专门为 FFT 运算提供的。

下面是一个 8～256 点基 2 复数 FFT 的 TMS320C2xx 汇编程序。

【例 14.2】8～256 点基 2 复数 FFT 的 TMS320C2xx 汇编程序

程序说明：

（1）数据页指针 DP=0

（2）用到的寄存器：AR0,AR1,AR2,AR3

（3）输入参数：N，M，WKAD，NOM。

其中，N=FFT 的点数，M=LOG2（N）；

WKAD 为系数的存放地址；

32 点 FFT：N=32，M=5，WKAD=WK32；

64 点 FFT：N=64，M=6，WKAD=WK64；

128 点 FFT：N=128，M=7，WKAD=WK128；

256 点 FFT：N=256，M=8，WKAD=WK256；

NOM：控制 FFT 运算是是否进行归一化

（4）调用 FFT 子程序前输入数据已经存放在 INDATR,INDATI，存放顺序为：

实部：在 INDATR，按顺序存放

虚部：在 INDATI， 按顺序存放

```
            .global  _main
INPUT:      .equ     200h        ; FFT 运算时输入数据的位置
INDATR:     .equ     2000h       ; 输入数据实部存放地址
INDATI:     .equ     2100h       ; 输入数据虚部存放地址
SINTAB:     .equ     2200h       ; 系数表在内存中的地址
;
;内部存储器定义：
N:          .equ     118         ; FFT 点数
M:          .equ     119         ; pow（2,M）=N
WKAD:       .equ     120         ; 系数的起始地址
NOM:        .equ     121         ; 归一化（0）,不归一化（1）
IW:         .equ     122         ; 系数增量
XT:         .equ     123         ; 暂存
YT:         .equ     124         ; 暂存
ID:         .equ     125         ; DISTANT BETWEEN 2 COMPUTING DATAS
C1:         .equ     126         ; 循环计数器
C2:         .equ     127         ; 循环计数器
      .text
_main:
          LDPK     0
```

```
        LACK    256
        SACL    N
        LACK    8
        SACL    M
        LACK    0
        SACL    NOM
        LALK    WK256
        LRLK    AR1,SINTAB
        RPTK    255
        TBLR    *+                  ; 将系数表调入 SINTAB 开始的 RAM 中
        CALL    FFT                 ; 作 FFT 运算
HERE    B       HERE

        ; 以下是 FFT 子程序
FFT     LDPK    0
        LAC     M
        SUBK    1
        SACL    M                   ; M=M－1
        LARP    AR1
        LALK    SINTAB
        SACL    WKAD
FFT1    LAR     AR0,N
        LAC     N
        SUBK    1
        SACL    N                   ; N=N－1
        LARP    AR1
        ; 将输入数据调入片内 RAM，存放顺序为：实部、虚部；实部、虚部…。
        LRLK    AR2,INPUT
        LRLK    AR1,INDATR
        LAR     AR3,N
MF1     LAC     *+,0,AR2
        SACL    *BR0+,0,AR3
        BANZ    MF1,*-,AR1
        LRLK    AR1,INDATI
        LAR     AR3,N
MF2     LAC     *+,0,AR2
        SACL    *BR0+,0,AR3
        BANZ    MF2,*-,AR1
        SOVM
        SSXM                        ; 符号扩展
        SPM     1                   ; 结果左移一位
        LACK    1
        SACL    ID                  ; ID = 1
        LAC     N
        ADDK    1
        SACL    IW,1                ; IW = 2*N
        LAR     AR2,M
```

```
LOOP3   LRLK    AR1,INPUT          ; AR1=INPUT- → P.X
        LAC     ID,1
        SACL    ID                 ; ID = 2*ID = 2
        LAC     IW,15
        SACH    IW                 ; IW= IW /2 = N
        LAC     IW,15
        SACH    C2                 ; C2 = IW/2 = N/2
        LAR     AR0,ID
LOOP2   LAR     AR3,WKAD           ; AR3 指向 WK 的起始地址
        LAC     ID,15
        SACH    C1                 ; C1 = ID/2 = 1
        MAR     *0+,AR3            ; AR1= AR1 + D- → Q.X
LOOP1 ZAC
        LT      *+,AR1
        MPY     *+,AR3
        LT      *,AR1
        MPYA    *－,AR3
        SPAC
        SACH    XT                 ; XT = Q.X*COS-Q.Y*SIN
        ZAC
        LT      *－,AR1
        MPY     *+,AR3
        LT      *,AR1
        MPYA    *－
        APAC
        SACH    YT                 ; YT = Q.X*SIN+Q.Y*COS

        LAC     NOM
        BNZ     D2                 ; IF NOM=1 归一化
********* A_BEGIN ************     ; else 不归一化
        MAR     *0-                ; （AR1） = P.X
        LAC     *
        ADD     XT
        SACL    *0+                ; P.X = XT+P.X
        SUB     XT,1
        SACL    *+                 ; Q.X = P.X-XT; （AR1）=Q.Y
        MAR     *0-                ; （AR1） = P.Y
        LAC     *
        ADD     YT
        SACL    *0+                ; P.Y=P.Y+YT
        SUB     YT,1
        SACL    *+,0,AR3           ; Q.Y=P.Y-YT
        B       D                  ; （AR1）=Q.Y+1- → NEXT Q.X
******** A_END************
******** B_BEGIN **********
D2      MAR     *0-                ; （AR1）=P.X
        LAC     *,15
```

```
        ADD     XT,15
        SACH    *0+             ; P.X=（XT+P.X）/2
        SUBH    XT
        SACH    *+              ; Q.X=（P.X-XT）/2;（AR1）=Q.Y
        MAR     *0-             ; （AR1）=P.Y
        LAC     *,15
        ADD     YT,15
        SACH    *0+             ; P.Y=（P.Y+YT）/2
        SUBH    YT
        SACH    *+,0,AR3        ; Q.Y=（P.Y-YT）/2
                                ; （AR1）=Q.Y+1- → NEXT Q.X
******  B_END ******
D       LAR     AR0,IW
        MAR     *0+             ; （AR3）=（AR3）+IW- → WK.COS
        LAR     AR0,ID          ; （AR0）=ID
        LAC     C1
        SUBK    1
        SACL    C1              ; C1=C1-1
        BGZ     LOOP1
        LAC     C2
        SUBK    1
        SACL    C2              ; C2=C2-1
        BGZ     LOOP2,*,AR1
        LARP    AR2
        BANZ    LOOP3,*-,AR1
        RET
```

上述程序中，当 NOM 为 1 时，FFT 运算进行归一化，否则不进行归一化。其中，A_BEGIN 到 A_END 之间的程序为不需要归一化的程序，而 B_BEGIN 到 B_END 之间的程序为需要归一化的程序。FFT 的运算结果在 200h～（200h+2*N-1）的片内 RAM 中，存放顺序为：实部、虚部，实部、虚部……

以下是 8 点～256 点 FFT 系数表，存放顺序为 cos,sin; cos,sin; …。

```
        .data
WK8:    .equ    $
        .word   07fffh,00h,05a82h,0a57eh,00h,08001h,0a57eh,0a57eh
WK16:   .equ    $
        .word   07fffh,00h,07642h,0cf05h,05a82h,0a57eh,030fch,089bfh
        .word   00h,08001h,0cf04h,089bfh,0a57eh,0a57eh,089beh,0cf05h
WK32:   .equ    $
        .word   07fffh,00h,07d8ah,0e708h,07642h,0cf05h,06a6eh,0b8e4h
        .word   05a82h,0a57eh,0471dh,09593h,030fch,089bfh,018f9h,08276h
        .word   00h,08001h,0e707h,08276h,0cf04h,089bfh,0b8e3h,09593h
        .word   0a57eh,0a57eh,09592h,0b8e4h,089beh,0cf05h,08276h,0e708h
WK64:   .equ    $
        .word   07fffh,00h,07f62h,0f375h,07d8ah,0e708h,07a7dh,0dad8h
        .word   07642h,0cf05h,070e3h,0c3aah,06a6eh,0b8e4h,062f2h,0aecdh
        .word   05a82h,0a57eh,05134h,09d0eh,0471dh,09593h,03c57h,08f1eh
```

```
        .word   030fch,089bfh,02528h,08583h,018f9h,08276h,0c8ch,0809eh
        .word   00h,08001h,0f374h,0809eh,0e707h,08276h,0dad8h,08583h
        .word   0cf04h,089bfh,0c3a9h,08f1eh,0b8e3h,09593h,0aecch,09d0eh
        .word   0a57eh,0a57eh,09d0eh,0aecdh,09592h,0b8e4h,08f1dh,0c3aah
        .word   089beh,0cf05h,08583h,0dad8h,08276h,0e708h,0809eh,0f375h
WK128: .equ   $
        .word   07fffh,00h,07fd9h,0f9b9h,07f62h,0f375h,07e9dh,0ed38h
        .word   07d8ah,0e708h,07c2ah,0e0e7h,07a7dh,0dad8h,07885h,0d4e1h
        .word   07642h,0cf05h,073b6h,0c946h,070e3h,0c3aah,06dcah,0be32h
        .word   06a6eh,0b8e4h,066d0h,0b3c1h,062f2h,0aecdh,05ed7h,0aa0bh
        .word   05a82h,0a57eh,055f6h,0a129h,05134h,09d0eh,04c40h,09931h
        .word   0471dh,09593h,041ceh,09236h,03c57h,08f1eh,036bah,08c4bh
        .word   030fch,089bfh,02b1fh,0877ch,02528h,08583h,01f1ah,083d7h
        .word   018f9h,08276h,012c8h,08163h,0c8ch,0809eh,0648h,08028h
        .word   00h,08001h,0f9b8h,08028h,0f374h,0809eh,0ed38h,08163h
        .word   0e707h,08276h,0e0e6h,083d7h,0dad8h,08583h,0d4e1h,0877ch
        .word   0cf04h,089bfh,0c946h,08c4bh,0c3a9h,08f1eh,0be32h,09236h
        .word   0b8e3h,09593h,0b3c0h,09931h,0aecch,09d0eh,0aa0ah,0a129h
        .word   0a57eh,0a57eh,0a129h,0aa0bh,09d0eh,0aecdh,09930h,0b3c1h
        .word   09592h,0b8e4h,09236h,0be32h,08f1dh,0c3aah,08c4ah,0c946h
        .word   089beh,0cf05h,0877bh,0d4e1h,08583h,0dad8h,083d6h,0e0e7h
        .word   08276h,0e708h,08163h,0ed38h,0809eh,0f375h,08027h,0f9b9h
WK256:  .equ    $
        .word   07fffh,00h,07ff6h,0fcdch,07fd9h,0f9b9h,07fa7h,0f696h
        .word   07f62h,0f375h,07f0ah,0f055h,07e9dh,0ed38h,07e1eh,0ea1eh
        .word   07d8ah,0e708h,07ce4h,0e3f5h,07c2ah,0e0e7h,07b5dh,0ddddh
        .word   07a7dh,0dad8h,0798ah,0d7dah,07885h,0d4e1h,0776ch,0d1efh
        .word   07642h,0cf05h,07505h,0cc22h,073b6h,0c946h,07255h,0c674h
        .word   070e3h,0c3aah,06f5fh,0c0e9h,06dcah,0be32h,06c24h,0bb86h
        .word   06a6eh,0b8e4h,068a7h,0b64ch,066d0h,0b3c1h,064e9h,0b141h
        .word   062f2h,0aecdh,060ech,0ac65h,05ed7h,0aa0bh,05cb4h,0a7beh
        .word   05a82h,0a57eh,05843h,0a34ch,055f6h,0a129h,0539bh,09f14h
        .word   05134h,09d0eh,04ec0h,09b18h,04c40h,09931h,049b4h,0975ah
        .word   0471dh,09593h,0447bh,093dch,041ceh,09236h,03f17h,090a1h
        .word   03c57h,08f1eh,0398dh,08dabh,036bah,08c4bh,033dfh,08afch
        .word   030fch,089bfh,02e11h,08894h,02b1fh,0877ch,02827h,08676h
        .word   02528h,08583h,02224h,084a3h,01f1ah,083d7h,01c0ch,0831dh
        .word   018f9h,08276h,015e2h,081e3h,012c8h,08163h,0fabh,080f7h
        .word   0c8ch,0809eh,096bh,08059h,0648h,08028h,0324h,0800ah
        .word   00h,08001h,0fcdch,0800ah,0f9b8h,08028h,0f695h,08059h
        .word   0f374h,0809eh,0f055h,080f7h,0ed38h,08163h,0ea1eh,081e3h
        .word   0e707h,08276h,0e3f4h,0831dh,0e0e6h,083d7h,0dddch,084a3h
        .word   0dad8h,08583h,0d7d9h,08676h,0d4e1h,0877ch,0d1efh,08894h
        .word   0cf04h,089bfh,0cc21h,08afch,0c946h,08c4bh,0c673h,08dabh
        .word   0c3a9h,08f1eh,0c0e9h,090a1h,0be32h,09236h,0bb85h,093dch
        .word   0b8e3h,09593h,0b64ch,0975ah,0b3c0h,09931h,0b140h,09b18h
        .word   0aecch,09d0eh,0ac65h,09f14h,0aa0ah,0a129h,0a7bdh,0a34ch
```

```
.word   0a57eh,0a57eh,0a34ch,0a7beh,0a129h,0aa0bh,09f14h,0ac65h
.word   09d0eh,0aecdh,09b17h,0b141h,09930h,0b3c1h,09759h,0b64ch
.word   09592h,0b8e4h,093dch,0bb86h,09236h,0be32h,090a1h,0c0e9h
.word   08f1dh,0c3aah,08dabh,0c674h,08c4ah,0c946h,08afbh,0cc22h
.word   089beh,0cf05h,08894h,0d1efh,0877bh,0d4e1h,08676h,0d7dah
.word   08583h,0dad8h,084a3h,0ddddh,083d6h,0e0e7h,0831ch,0e3f5h
.word   08276h,0e708h,081e2h,0ea1eh,08163h,0ed38h,080f6h,0f055h
.word   0809eh,0f375h,08059h,0f696h,08027h,0f9b9h,0800ah,0fcdch
```



14.4.3　FFT 的 TMS320C54x 实现

下面是一个 16 ~1024 点基 2 实数 FFT 的 TMS320C54x 汇编程序。2N 点实数 FFT 采用 N 点复数 FFT 实现，包括以下四个步骤：打包和比特反转、N 点复数 FFT、分离奇共轭和偶共轭分量、结果生成，这里打包就是将相邻的两个实数当做一个复数。

【例 14.3】16 ~1024 点基 2 实数 FFT 的 TMS320C54x 汇编程序

程序主体说明：程序由 4 个子程序组成：

（1）rfft_task：主调用子程序，它调用算法的其他子程序，实现统一的接口。

（2）bit_rev：实现输入数据的比特反转。

（3）fft：实现 N 点复数 FFT，FFT 变换以同址的方式完成。在运算的过程中，为了避免运算结果的溢出，对每个蝶形的运算结果右移一位。因为第一级和第二级蝶形运算不需要乘法，所以 fft 的实现又可以分为三个功能块：第一级蝶形运算、第二级蝶形运算、第三级至 logN 级蝶形运算。

（4）unpack：fft 运算的结果可以看做将原 2N 点实数的奇地址数据作 N 点复数的实部 R，偶地址数据作 N 点复数的虚部 I。利用 FFT 的对称性，可以由 N 点复数的 FFT 结果求 R 和 I 的 FFT 结果。再利用原 2N 点实数的 FFT 变换和上述结果的关系得到最后的结果。

```
;--------------------------------
;rfft_task
;--------------------------------
        K_DATA_IDX_1   .set     2               ; 第一级运算时各组蝶形的地址增量
        K_DATA_IDX_2   .set     4               ; 第二级运算时各组蝶形的地址增量
        K_DATA_IDX_3   .set     8               ; 第三级运算时各组蝶形的地址增量
        K_FLY_COUNT_3 .set      4               ; 第三级运算时每组蝶形的蝶形数
        .def     rfft_task
        .asg     AR1,FFT_TWID_P
        .sect    "rfft_prg"
 rfft_task:
        STM      #sine,FFT_TWID_P
        RPT      #K_FFT_SIZE-1                  ; 将正弦表从程序区搬移到数据区
        MVPD     #sine_table,*FFT_TWID_P+
        STM      #cosine,FFT_TWID_P
        RPT      #K_FFT_SIZE-1                  ; 将余弦表从程序区搬移到数据区
        MVPD     #cos_table,*FFT_TWID_P+
        CALL     bit_rev
        CALL     fft
        CALL     unpack
```

```
        RET
;-------------------------------------
;               .def    bit_rev
;-------------------------------------
        .asg    AR2,REORDERED_DATA
        .asg    AR3,ORIGINAL_INPUT
        .asg    AR7,DATA_PROC_BUF
        .sect   "rfft_prg"
bit_rev:
        SSBX    FRCT                ; FRCT=1
        MVDK    *（d_input_addr）,ORIGINAL_INPUT    ; AR3 指向输入数据块的首地址
                STM     #fft_data,DATA_PROC_BUF     ; AR7 指向数据处理缓冲区首地址
                MVMM    DATA_PROC_BUF,REORDERED_DATA
                                                    ; AR2 指向比特反转后数据首地址
                STM     #K_FFT_SIZE-1,BRC
                RPTBD   bit_rev_end-1
                STM     #K_FFT_SIZE,AR0             ; AR0 = 缓冲区大小的 1/2
                MVDD    *ORIGINAL_INPUT+,*REORDERED_DATA+
                MVDD    *ORIGINAL_INPUT-,*REORDERED_DATA+
                MAR     *ORIGINAL_INPUT+0B
bit_rev_end:
                RET                                 ; 返回实数 FFT 主模块
;-----------------------------
;               fft
;-----------------------------
                .asg    AR1,GROUP_COUNTER
                .asg    AR2,PX
                .asg    AR3,QX
                .asg    AR4,WR
                .asg    AR5,WI
                .asg    AR6,BUTTERFLY_COUNTER
                .asg    AR7,DATA_PROC_BUF       ; for Stages 1 & 2
                .asg    AR7,STAGE_COUNTER       ; for the remaining stages
                .sect   "rfft_prg"
fft:
; Stage 1
                STM     #0,BK                   ; BK=0 so that *ARn+0% =*ARn+0
                LD      #-1,ASM                 ; 每级输出除以 2，避免溢出
                MVMM    DATA_PROC_BUF,PX        ; PX → PR
                LD      *PX,16,A                ; A := PR
                STM     #fft_data+K_DATA_IDX_1,QX   ; QX → QR
                STM     #K_FFT_SIZE/2-1,BRC
                RPTBD   stage1end-1
                STM     #K_DATA_IDX_1+1,AR0
                SUB     *QX,16,A,B              ; B := PR-QR
                ADD     *QX,16,A                ; A := PR+QR
                STH     A,ASM,*PX+              ; PR':= （PR+QR）/2
```

```
                ST      B,*QX+                  ; QR' :=  (PR-QR) /2
                ||LD    *PX,A                   ; A := PI
                SUB     *QX,16,A,B              ; B := PI-QI
                ADD     *QX,16,A                ; A := PI+QI
                STH     A,ASM,*PX+0             ; PI':=  (PI+QI) /2
                ST      B,*QX+0%                ; QI':= (PI+QI) /2
                ||LD    *PX,A                   ; A := next PR
stage1end:
; Stage 2
        MVMM    DATA_PROC_BUF,PX                ; PX → PR
        STM     #fft_data+K_DATA_IDX_2,QX       ; QX → QR
        STM     #K_FFT_SIZE/4-1,BRC
        LD      *PX,16,A                        ; A := PR
        RPTBD   stage2end-1
            STM     #K_DATA_IDX_2+1,AR0
; 第 1 个蝶形
        SUB     *QX,16,A,B                      ; B := PR-QR
        ADD     *QX,16,A                        ; A := PR+QR
        STH     A,ASM,*PX+                      ; PR':=  (PR+QR) /2
        ST      B,*QX+                          ; QR':=  (PR-QR) /2
        ||LD    *PX,A                           ; A := PI
        SUB     *QX,16,A,B                      ; B := PI-QI
        ADD     *QX,16,A                        ; A := PI+QI
        STH     A,ASM,*PX+                      ; PI':=  (PI+QI) /2
        STH     B,ASM,*QX+                      ; QI':=  (PI-QI) /2
; 第 2 个蝶形
        MAR     *QX+
        ADD     *PX,*QX,A                       ; A := PR+QI
        SUB     *PX,*QX-,B                      ; B := PR-QI
        STH     A,ASM,*PX+                      ; PR':=  (PR+QI) /2
        SUB     *PX,*QX,A                       ; A := PI-QR
        ST      B,*QX                           ; QR':=  (PR-QI) /2
        ||LD    *QX+,B                          ; B := QR
        ST      A, *PX                          ; PI:=  (PI-QR) /2
        ||ADD   *PX+0%,A                        ; A := PI+QR
        ST      A,*QX+0%                        ; QI':=  (PI+QR) /2
        ||LD    *PX,A                           ; A := PR
stage2end:
; Stage 3 到 Stage logN–1
        STM     #K_TWID_TBL_SIZE,BK             ; BK = twiddle table size always
        ST      #K_TWID_IDX_3,*(d_twid_idx)     ; init index of twiddle table
        STM     #K_TWID_IDX_3,AR0               ; AR0 = index of twiddle table
        STM     #cosine,WR                      ; init WR pointer
        STM     #sine,WI                        ; init WI pointer
        STM     #K_LOGN-2-1,STAGE_COUNTER       ; init stage counter
        ST      #K_FFT_SIZE/8-1,*(d_grps_cnt)   ; init group counter
        STM     #K_FLY_COUNT_3-1,BUTTERFLY_COUNTER;init butterfly counter
```

```
        ST      #K_DATA_IDX_3,*（d_data_idx）     ; init index for input data
stage:
        STM     #fft_data,PX                ; PX → PR
        LD      *（d_data_idx）, A
        ADD     *（PX）,A
        STLM    A,QX                        ; QX → QR
        MVDK    *（d_grps_cnt）,GROUP_COUNTER   ; AR1 contains group counter
group:
        MVMD    BUTTERFLY_COUNTER,BRC   ; # of butterflies in each group
        RPTBD   butterflyend-1
        LD      *WR,T                   ; T := WR
        MPY     *QX+,A                  ; A := QR*WR || QX → QI
        MACR    *WI+0%,*QX-,A           ; A := QR*WR+QI*WI
                                        ; || QX → QR
        ADD     *PX,16,A,B              ; B :=  （QR*WR+QI*WI）+PR
        ST      B,*PX                   ; PR':=（（QR*WR+QI*WI）+PR）/2
        ||SUB   *PX+,B                  ; B := PR-（QR*WR+QI*WI）
                                        ; || PX → PI
        ST      B,*QX                   ; QR':=  （PR-（QR*WR+QI*WI））/2
        ||MPY   *QX+,A                  ; A := QR*WI [T=WI]
                                        ; || QX → QI
        MASR    *QX,*WR+0%,A            ; A := QR*WI-QI*WR
        ADD     *PX,16,A,B              ; B :=  （QR*WI-QI*WR）+PI
        ST      B,*QX+                  ; QI':=（（QR*WI-QI*WR）+PI）/2
                                        ; || QX → QR
        ||SUB   *PX,B                   ; B := PI-（QR*WI-QI*WR）
        LD      *WR,T                   ; T := WR
        ST      B,*PX+                  ; PI':=  （PI-（QR*WI-QI*WR））/2
                                        ; || PX → PR
        ||MPY   *QX+,A                  ; A := QR*WR || QX → QI
butterflyend:
        ; 为下一组更新指针
        PSHM    AR0                     ; preserve AR0
        MVDK    *（d_data_idx）,AR0
        MAR     *PX+0                   ; increment PX for next group
        MAR     *QX+0                   ; increment QX for next group
        BANZD   group,*GROUP_COUNTER-
        POPM    AR0                    ; restore AR0
        MAR     *QX-                   ; Update counters and indices for next stage
        LD      *（d_data_idx）,A
        SUB     #1,A,B                  ; B = A-1
        STLM    B,BUTTERFLY_COUNTER; BUTTERFLY_COUNTER = #flies–1
        STL     A,1,*（d_data_idx）     ; double the index of data
        LD      *（d_grps_cnt）,A
        STL     A,ASM,*（d_grps_cnt）   ; 1/2 the offset to next group
        LD      *（d_twid_idx）,A
        STL     A,ASM,*（d_twid_idx）   ; 1/2 the index of twiddle table
```

```
        MVDK    *（d_twid_idx）,AR0        ; AR0 = index of twiddle table
        BANZ    stage,*STAGE_COUNTER-
fft_end:
        RET                                          ; 返回实数 FFT 主模块
;-----------------------------------
;unpack
;-----------------------------------
        .sect "rfft_prg"
unpack:
; 计算中间值 RP, RM, IP, IM
        .asg    AR2,XP_k
        .asg    AR3,XP_Nminusk
        .asg    AR6,XM_k
        .asg    AR7,XM_Nminusk
        STM             #fft_data+2,XP_k            ; AR2 → R[k] （temp RP[k]）
        STM             #fft_data+2*K_FFT_SIZE-2,XP_Nminusk
                                                    ; AR–> R[N–k] （teRP[N–k]）mp
        STM             #fft_data+2*K_FFT_SIZE+3,XM_Nminusk  ; AR7 –> temp RM[N–k]
        STM             #fft_data+4*K_FFT_SIZE-1,XM_k        ; AR6 –> temp RM[k]
        STM             #-2+K_FFT_SIZE/2,BRC
        RPTBD           phase3end-1
        STM             #3,AR0
        ADD             *XP_k,*XP_Nminusk,A         ; A := R[k]+R[N–k] =2*RP[k]
        SUB             *XP_k,*XP_Nminusk,B         ; B := R[k]–R[N–k] 2*RM[k]=
        STH             A,ASM,*XP_k+                    ; store RP[k] at AR[k]
        STH             A,ASM,*XP_Nminusk+              ; store RP[N–k]=RP[k] atAR[N–k]
        STH             B,ASM,*XM_k-                    ; store RM[k] at AI[2N–k]
        NEG             B                               ; B := R[N–k]–R[k] =2*RM[N–k]
        STH             B,ASM,*XM_Nminusk-              ; store RM[N–k] at AI[N+k]
        ADD             *XP_k,*XP_Nminusk,A         ; A := I[k]+I[N–k] =2*IP[k]
        SUB             *XP_k,*XP_Nminusk,B         ; B := I[k]–I[N–k] 2*IM[k]=
        STH             A,ASM,*XP_k+                    ; store IP[k] at AI[k]
        STH             A,ASM,*XP_Nminusk-0         ; store IP[N–k]=IP[k] atAI[N–k]
        STH             B,ASM,*XM_k-                    ; store IM[k] at AR[2N–k]
        NEG             B                               ; B := I[N–k]–I[k] =2*IM[N–k]
        STH             B,ASM,*XM_Nminusk+0         ; store IM[N–k] at AR[N+k]
phase3end:
        ST          #0,*XM_k-                       ; RM[N/2]=0
        ST          #0,*XM_k                        ; IM[N/2]=0
; 计算 AR[0],AI[0], AR[N], AI[N]
        .asg    AR2,AX_k
        .asg    AR4,IP_0
        .asg    AR5,AX_N
        STM             #fft_data,AX_k              ; AR2 –> AR[0] （tempRP[0]）
        STM             #fft_data+1,IP_0            ; AR4 –> AI[0] （tempIP[0]）
        STM             #fft_data+2*K_FFT_SIZE+1,AX_N   ; AR5 –> AI[N]
        ADD             *AX_k,*IP_0,A               ; A := RP[0]+IP[0]
```

```
        SUB     *AX_k,*IP_0,B                   ; B := RP[0]–IP[0]
        STH     A,ASM,*AX_k+                    ; AR[0] = （RP[0]+IP[0]）/2
        ST      #0,*AX_k                        ; AI[0] = 0
        MVDD    *AX_k+,*AX_N-                   ; AI[N] = 0
        STH     B,ASM,*AX_N                     ; AR[N] = （RP[0]–IP[0]）/2
; 计算最终输出值 AR[k], AI[k]
        .asg    AR3,AX_2Nminusk
        .asg    AR4,COS
        .asg    AR5,SIN
        STM     #fft_data+4*K_FFT_SIZE-1,AX_2Nminusk
                                                ; AR3–>AI[2N–1（temp RM[1]）]
        STM     #cosine+K_TWID_TBL_SIZE/K_FFT_SIZE,COS
                                                ; AR4 →cos（k*pi/N）
        STM     #sine+K_TWID_TBL_SIZE/K_FFT_SIZE,SIN    ; AR5 –> sin（k*pi/N）
        STM     #K_FFT_SIZE-2,BRC
        RPTBD   phase4end-1
        STM     #K_TWID_TBL_SIZE/K_FFT_SIZE,AR0
; index of twiddle tables
        LD      *AX_k+,16,A             ; A := RP[k] || AR2–>IP[k]
        MACR    *COS,*AX_k,A            ; A :=A+cos（k*pi/N）*IP[k]
        MASR    *SIN,*AX_2Nminusk-,A    ; A := A–sin（k*pi/N*RM[k]）
        LD      *AX_2Nminusk+,16,B      ; B := IM[k] ||AR3–>RM[k]
        MASR    *SIN+0%,*AX_k-,B        ; B := B–sin（k*pi/*IP[k]N）
        MASR    *COS+0%,*AX_2Nminusk,B  ; B := B–cos（k*pi/N）*RM[k]
        STH     A,ASM,*AX_k+            ; AR[k] = A/2
        STH     B,ASM,*AX_k+            ; AI[k] = B/2
        NEG     B                       ; B := –B
        STH     B,ASM,*AX_2Nminusk-     ; AI[2N–k] = –AI[= B/2k]
        STH     A,ASM,*AX_2Nminusk-     ; AR[2N–k] = A[k] = A/2R
phase4end:
        RET                             ; 返回实数 FFT 主模块
        .end
```

程序说明：

（1）正弦和余弦表。正弦和余弦表分别存放 M 点数据，各点值为 sin（k*pi/M） 和 cos（k*pi/M），k=0～（M-1），采用 Q15 表示。因为对正余弦表采用了变化增量的循环寻址，只要 M≥N/2，即可用同一个正余弦表实现点数小于 N 的 FFT 变换。

（2）数据存放要求。因为采用了循环寻址，数据区的 M 点 sin 表和 M 点 cos 表必须放于 q 位地址边界 （2^q>M）。因为采用了比特反转寻址，输入 2N 点实数要放于 m 位地址边界，2^m≥2N。并且，原输入数据区要作为 FFT 结果数据区的下半部分，以实现同址运算。

（3）常量说明。K_FFT_SIZE, K_LOGN, K_TWID_TBL_SIZE 分别表示把输入数据当复数时的点数、复数 FFT 的级数、正余弦表的大小。K_TWID_IDX_3 表示第三级以循环寻址方式对正余弦表寻址时的增量 M/4，例如，当正弦和余弦表为 128 点时，K_TWID_IDX_3 取 32。

（4）地址和变量说明。sine_table、cos_table 为存放于数据区的正余弦表的首地址。sine、cosine 为存放于程序区的正余弦表的首地址。fft_data 为 FFT 运算和结果数据首地址。

d_input_addr 用于存放输入数据首地址。d_grps_cnt 用于存放每级运算时蝶形组数。d_data_idx 用于存放每组蝶形间的地址增量、d_twid_idx 用于存放循环寻址对正余弦表寻址时的增量。

14.5　FFT 的浮点 DSP 实现

与定点 DSP 芯片相比，用浮点 DSP 芯片实现 FFT 算法可以不考虑数据的溢出。下面我们以 TMS320C3x 为例来考虑 FFT 算法的浮点 DSP 芯片实现。

与定点 DSP 芯片一样，为了提高 FFT 算法的运算速度，实现 FFT 运算时特别需要考虑如下两点：

（1）利用片内 RAM。由于片内 RAM 具有双寻址特性，因此，采用片内 RAM 存放数据实现 FFT 算法要比采用外部全速 RAM 还要快。这是因为，采用外部 RAM 一方面在一个周期内只能作一次寻址，另一方面，还可能产生流水线冲突，造成插入额外的周期。

（2）利用比特反转寻址。TMS320C3x 提供了专用于 FFT 运算的比特反转寻址方式。

此外，如果输入数据是以连续形式输入的，那么采用 DMA 功能可以减少数据输入存储的时间。例如，从串行口输入数据时，DMA 的源地址就指向串行口，而 DMA 的目的地址则指向数据的存储空间。

下面我们给出两个基 2 FFT 运算的 TMS320C3x 汇编程序。其中，例 14.4 是基 2 复数 FFT 程序，例 14.5 是基 2 实数 FFT 程序，这两个程序都可以被 C 程序所调用。

【例 14.4】TMS320C3x 复数基 2 DIF FFT 程序（C 程序可调用）

```
* 函数调用时的堆栈结构:
*                  +-------------------+
*     -FP（4）     |       data        |
*     -FP（3）     |         M         |
*     -FP（2）     |         N         |
*     -FP（1）     |       返回地址    |
*     -FP（0）     |      旧 FP        |
*                  +-------------------+
        .globl     _fft_compx
        .globl     _sine
FP      .set       AR3
        .bss       FFTSIZ,1                ; FFT 点数
        .bss       LOGFFT,1                ; FFT 级数
        .bss       INPUT,1                 ; 数组地址
        .text
SINTAB.word        _sine
_fft_compx:
        PUSH       FP                      ; 保护寄存器
        LDI        SP,FP
        PUSH       R4
        PUSH       R5
        PUSHF      R6
        PUSHF      R7
        PUSH       AR4
```

```
        PUSH    AR5
        PUSH    AR6
        PUSH    AR7
        LDI     *-FP（2）,R0
        STI     R0,@FFTSIZ
        LDI     *-FP（3）,R0
        STI     R0,@LOGFFT
        LDI     *-FP（4）,R0
        STI     R0,@INPUT
        LDI     @FFTSIZ,IR1
        LSH     -2,IR1                  ; IR1=N/4
        LDI     0,AR6                   ; AR6 内含第几级运算
        LDI     @FFTSIZ,IR0
        LSH     1,IR0
        LDI     @FFTSIZ,R7
        LDI     1,AR7
        LDI     1,AR5
;外部循环
LOOP: NOP       *++AR6（1）
        LDI     @INPUT,AR0              ; AR0 指向 X（I）
        ADDI    R7,AR0,AR2              ; AR2 指向 X（L）
        LDI     AR7,RC
        SUBI    1,RC
;第一个循环
        RPTB    BLK1
        ADDF    *AR0,*AR2,R0        ; R0=X（I）+X（L）
        SUBF    *AR2++,*AR0++,R1    ; R1=X（I）−X（L）
        ADDF    *AR2,*AR0,R2        ; R2=Y（I）+Y（L）
        SUBF    *AR2,*AR0,R3        ; R3=Y（I）−Y（L）
        STF     R2,*AR0--           ; Y（I）=R2
       ||STF    R3,*AR2--           ; Y（L）=R3
BLK1: STF       R0,*AR0++（IR0）    ; X（I）=R0
       ||STF    R1,*AR2++（IR0）    ; X（L）=R1
;如果是最后一级，则 FFT 结束
        CMPI    @LOGFFT,AR6
        BZD     END
;内部主循环
        LDI     2,AR1                   ; 内循环的循环计数器
        LDI     @SINTAB,AR4
INLOP:ADDI      AR5,AR4
        LDI     AR1,AR0
        ADDI    2,AR1
        ADDI    @INPUT,AR0
        ADDI    R7,AR0,AR2
        LDI     AR7,RC
        SUBI1,RC
        LDF     *AR4,R6                 ; R6=SIN
```

```
;第二个循环
        RPTB    BLK2
        SUBF    *AR2,*AR0,R2        ; R2=X（I）−X（L）
        SUBF    *+AR2,*+AR0,R1      ; R1=Y（I）-Y（L）
        MPYF    R2,R6,R0            ; R0=R2*SIN
        ||ADDF  *+AR2,*+AR0,R3      ; R3=Y（I）+Y（L）
        MPYF    R1,*+AR4（IR1）,R3; R3=R1*COS
        ||STF   R3,*+AR0            ; Y（I）=Y（I）+Y（L）
        SUBF    R0,R3,R4            ; R4=R1*COS-R2*SIN
        MPYF    R1,R6,R0            ; R0=R1*SIN
        ||ADDF  *AR2,*AR0,R3        ; R3=X（I）+X（L）
        MPYF    R2,*+AR4（IR1）,R3; R3=R2*COS
        ||STF   R3,*AR0++（IR0）    ; X（I）=X（I）+X（L）
        ADDF    R0,R3,R5            ; R5=R2*COS+R1*SIN
        BLK2：STF  R5,*AR2++（IR0）; X（L）=R2*COS+R1*SIN
        ||STF   R4,*+AR2            ; Y（L）=R1*COS-R2*SIN
        CMPI    R7,AR1
        BNE     INLOP                   ; 跳至内循环
        LSH     1,AR7
        BRD     LOOP                    ; 作下一级 FFT 运算
        LSH     1,AR5                   ; IE=2*IE
        LDI     R7,IR0                  ; N1=N2
        LSH     −1,R7                   ; N2=N2/2
;按变址方式存储结果
END:    LDI     @FFTSIZ,RC
        SUBI    1,RC                    ; RC=N−1
        LDI     @FFTSIZ,IR0             ; IR0=N
        LDI     2,IR1
        LDI     @INPUT,AR0
        LDI     @INPUT,AR1
        RPTB    BITRV
        LDF     *+AR0（1）,R0
        ||LDF   *AR0++（IR0）B,R1
BITRV   STF     R0,*+AR1（1）
        ||STF   R1,*AR1++（IR1）
;寄存器恢复
        POP     AR7
        POP     AR6
        POP     AR5
        POP     AR4
        POPF    R7
        POPF    R6
        POP     R5
        POP     R4
        POP     FP
        RETS
.end
```

调用上述程序时需要的正弦/余弦表在程序链接时应予提供。这个表格可以在 C 程序中用一全局数组表示如下：

```
float  sine[length]={      };
```

其中，length = 5*N/4，数组内的数值为 sin（i*2*pi/N），i 从 0 到 （5*N/4） -1。用 C 编译器编译可得：

```
_sine    .float  value1          = sin（0*2*pi/N）
         .float  value2          = sin（1*2*pi/N）
         ......
         .float value（5N/4）    = sin（（5*N/4-1）*2*pi/N）
```

调用 fft_complx FFT 程序的方法比较简单。根据 fft_complx 的定义，将 FFT 的点数 N、级数 M 及输入数组 data 传递给函数即可。例如要计算 256 点复数 FFT，首先将 256 点复数数据放入数组 data，数据的组织形式为实部和虚部交替存放，然后按如下方式调用即可：

```
fft_complx（256,8,data）;
```

【例 14.5】TMS320C3x 实数基 2 FFT 汇编程序（C 程序可调用）

```
* 函数调用时的堆栈结构:
*                +-------------------+
*     -FP（4）   |       data        |
*     -FP（3）   |        M          |
*     -FP（2）   |        N          |
*     -FP（1）   |    返回地址       |
*     -FP（0）   |     旧 FP         |
*                +-------------------+
FP      .set    AR3
        .global _fft_rl                 ; 程序入口点
        .global _sine                   ; 正弦表地址
        .bss    FFTSIZ,1                ; FFT 点数
        .bss    LOGFFT,1                ; FFT 级数
        .bss    INPUT,1                 ; 数组地址
        .text
SINTAB.word     _sine
_fft_rl: PUSH   FP                      ; 寄存器保护
        LDI     SP,FP
        PUSH    R4
        PUSH    R5
        PUSH    AR4
        PUSH    AR5
        LDI     *-FP（2）,R0                ; 参数传递：
        STI     R0,@FFTSIZ              ; FFTSIZ=N
        LDI     *-FP（3）,R0
        STI     R0,@LOGFFT              ; LOGFFT=M
        LDI     *-FP（4）,R0
        STI     R0,@INPUT               ; INPUT=数组 data 地址
;数据变址运算
        LDI     @FFTSIZ,RC              ; RC=N
        SUBI    1,RC                    ; RC=N-1
```

```
        LDI     @FFTSIZ,IR0
        LSH     -1,IR0                  ; IR0=N/2
        SUBI    R0,R0                   ; R0=0
        LDI     R0,AR0                  ; AR0=0
        LDI     R0,AR1                  ; AR1=0
        LDI     @INPUT,IR1              ;
        RPTB    BITRV
        CMPI    AR1,AR0                 ; 如果 AR0<AR1，交换位置
        BGE     CONT
        LDF     *+AR0（IR1）,R0
       ||LDF    *+AR1（IR1）,R1
        STF     R0,*+AR1（IR1）
       ||STF    R1,*+AR0（IR1）
CONT NOP        *AR0++
BITRV NOP       *AR1++（IR0）B
;    蝶形
        LDI     @INPUT,AR0              ; AR0 指向 X（I）
        LDI     IR0,RC                  ; 重复 N/2 次
        SUBI    1,RC                    ; RC=N/2 -1
        RPTB    BLK1
        ADDF    *+AR0,*AR0++,R0         ; R0=X（I）+X（I+1）
        SUBF    *AR0,*-AR0,R1           ; R1=X（I）-X（I+1）
BLK1  STF       R0,*-AR0                ; X（I）=X（I）+X（I+1）
       | STF    R1,*AR0++               ; X（I+1）=X（I）-X（I+1）
        LDI     @INPUT,AR0              ; AR0 指向 X（I）
        LDI     2,IR0                   ; IR0=2=N2
        LDI     @FFTSIZ,RC
        LSH     -2,RC                   ; 重复 N/4 次
        SUBI    1,RC
        RPTB    BLK2
        ADDF    *+AR0（IR0）,*AR0++（IR0）,R0    ;R0=X（I）+X（I+2）
        SUBF    *AR0,*-AR0（IR0）,R1    ; R1=X（I）-X（I+2）
        NEGF    *+AR0,R0                ; R0=-X（I+3）
       ||STF    R0,*-AR0（IR0）         ; X（I）=X（I）+X（I+2）
BLK2  STF       R1,*AR0++（IR0）        ; X（I+2）=X（I）-X（I+2）
       ||STF    R0,*+AR0                ; X（I+3）=-X（I+3）
; 主循环
        LDI     @FFTSIZ,IR0
        LSH     -2,IR0                  ; IR0=INDEX FOR E
        LDI     3,R5                    ; R5 holds the current stage number
        LDI     1,R4                    ; R4=N4
        LDI     2,R3                    ; R3=N2
LOOP  LSH       -1,IR0                  ; E=E/2
        LSH     1,R4                    ; N4=2*N4
        LSH     1,R3                    ; N2=2*N2
;内循环
        LDI     @INPUT,AR5              ; AR5 指向 X（I）
```

```
INLOP LDI     IR0,AR0
      ADDI    @SINTAB,AR0          ; AR0 指向 sin/cos 表
      LDI     R4,IR1               ; IR1=N4
      LDI     AR5,AR1
      ADDI    1,AR1                ; AR1 指向 X（I1）=X（I+J）
      LDI     AR1,AR3
      ADDI    R3,AR3               ; AR3 指向 X（I3）=X（I+J+N2）
      LDI     AR3,AR2
      SUBI    2,AR2                ; AR2 指向 X（I2）=X（I−J+N2）
      ADDI    R3,AR2,AR4           ; AR4 指向 X（I4）=X（I−J+N1）
      LDF     *AR5++（IR1）,R0     ; R0=X（I）
      ADDF    *+AR5（IR1）,R0,R1   ; R1=X（I）+X（I+N2）
      SUBF    R0,*++AR5（IR1）,R0  ; R0=-X（I）+X（I+N2）
      ||  STF R1,*-AR5（IR1）      ; X（I）=X（I）+X（I+N2）
      NEGF    R0                   ; R0=X（I）−X（I+N2）
      NEGF    *++AR5（IR1）,R1     ; R1=-X（I+N4+N2）
      ||  STF R0,*AR5              ; X（I+N2）=X（I）−X（I+N2）
      STF     R1,*AR5              ; X（I+N4+N2）=-X（I+N4+N2）
; 最内层的循环
      LDI     @FFTSIZ,IR1
      LSH     -2,IR1
      LDI     R4,RC
      SUBI    2,RC                 ;重复 N4−1 次
      RPTB    BLK3
      MPYF    *AR3,*+AR0（IR1）,R0 ; R0=X（I3）*COS
      MPYF    *AR4,*AR0,R1         ; R1=X（I4）*SIN
      MPYF    *AR4,*+AR0（IR1）,R1 ; R1=X（I4）*COS
      ||ADDF  R0,R1,R2             ; R2=X（I3）*COS+X（I4）*SIN
      MPYF    *AR3,*AR0++（IR0）,R0 ; R0=X（I3）*SIN
      SUBF    R0,R1,R0             ; R0=-X（I3）*SIN+X（I4）*COS !!!
      SUBF    *AR2,R0,R1           ; R1=-X（I2）+R0 !!!
      ADDF    *AR2,R0,R1           ; R1=X（I2）+R0 !!!
      ||STF   R1,*AR3++            ; X（I3）=-X（I2）+R0 !!!
      ADDF    *AR1,R2,R1           ; R1=X（I1）+R2
      ||STF   R1,*AR4--            ; X（I4）=X（I2）+R0 !!!
      SUBF    R2,*AR1,R1           ; R1=X（I1）−R2
      ||STF   R1,*AR1++            ; X（I1）=X（I1）+R2
BLK3  STF     R1,*AR2--            ; X（I2）=X（I1）−R2
      SUBI    @INPUT,AR5
      ADDI    R4,AR5               ; AR5=I+N1
      CMPI    @FFTSIZ,AR5
      BLTD    INLOP                ; 循环至最内层
      ADDI    @INPUT,AR5
      NOP
      NOP
      ADDI    1,R5
      CMPI    @LOGFFT,R5
```

```
        BLE     LOOP
; 寄存器恢复并返回
        POP     AR5
        POP     AR4
        POP     R5
        POP     R4
        POP     FP
        RETS
```

例 14.5 是一个 TMS320C3x 基 2 实数 FFT 程序，可以被 C 程序所调用。这个程序函数定义为 void fft_rl（N,M,data），其中 N 为 FFT 的点数，M 为 FFT 的级数，$N=2^M$，即 $M=\log_2(N)$。data 是输入和输出的数组。作为输入时，data 是长度为 N 的实数数组，按顺序排列。输出时，data 也包含 N 点数据，这 N 点数据中包含有 N/2+1 个 FFT 输出值。若用 R 和 I 分别表示输出的实部和虚部，则输出数据的排列为 R（0）,R（1）,…,R（N/2）,I（N/2-1）,…,I（1）。程序在开始时对输入数据作变址运算，因此如果输入时 data 已经是比特反转排列，则这段程序要省去。

蝶形因子所需的正弦/余弦表在程序链接时应予提供，且应有如下格式：

```
        .global     _sine
        .data
_sine   .float  value1    = sin（0*2*pi/N）
        .float  value2    = sin（1*2*pi/N）
        .....
        .float  value（N/2）   = cos（（N/4-1）*2*pi/N）
```

其中，从 value1 到 value（N/4） 对应于正弦波的第一个 1/4 周期，从 value（N/4+1） 到 value（N/2）对应于余弦波的第一个 1/4 周期。因此，实际上整个表就对应于正弦波的前半周期。与上例相同，正弦/余弦表也可用一个 C 数组表示。如果直接采用汇编语言，则可用如下程序产生实数 FFT 所需的正弦函数表 _sine（以 256 点 FFT 为例）：

```
#include <math.h>
#include <stdio.h>
#define N 256                                   //256 点 FFT
#define pi 3.141592654                          //定义π值
main（）
{
  FILE *fp;
  int          i;
  float     result;
  fp = fopen（"sintab.256", "w+"）;             // 生成的 sine 表存在 sintab.256 中
  fprintf（fp, "\n%s", "  .global _sine"）;     // 写入符号   .global    _sine
  fprintf（fp, "\n%s", "  .data"）;             // 写入符号    .data
  fprintf（fp "\n%s%7f", "_sine   .float   ", 0.0000000）; //第一点的正弦函数值
  for  （i = 1; n < N/2; n++）                  //计算 N/2-1 点正弦函数值
  {
    result = sin（i*2*pi/N）;                   //sine 表中的正弦函数值
    fprintf（fp, "\n%s%7f", "   .float   ", result）;        //写入文件
  }
```

```
    fclose（fp）;                                    //关闭文件
}
```

调用 fft_rl FFT 程序的方法与上例相同。根据 fft_rl 的定义，将 FFT 的点数 N、级数 M 及输入数组 data 传递给函数即可。例如要计算 256 点实数 FFT，首先将 256 点数据放入数组 data，然后按如下方式调用即可：

```
fft_rl（256,8,data）;
```

14.6 小　　结

本章我们讨论了 FFT 的基本原理，给出了 FFT 算法的 C 语言实现程序，并在此基础上介绍了 FFT 的定点和浮点 DSP 实现方法。本章所介绍的例子都是采用循环程序实现的基 2 FFT 算法，如果采用基 4 算法或直线式程序，那么算法的速度可以进一步提高，当然，速度的提高是通过增加存储量来换取的。

14.7 习题与思考题

（1）编写一个计算 256 点实数 FFT 的 TMS320C3x 程序并用 CCS 中的 Simulator 进行调试，要求 C 可调用。

（2）编写一个计算 256 点实数 FFT 的 TMS320C54x 程序并用 CCS 中的 Simulator 进行调试，要求 C 可调用。

附录 A　TMS320C54x 指令集

本附录按照指令的 4 种不同功能（算术、逻辑、程序控制、装载和存储），将助记符指令分类对照进行总结。表中，字/周期栏中，x/y[d]表示该指令代码字长为 x 个字，执行该指令需 y 个机器周期，若采用延时方式则需 d 个机器周期；x/y,z[d]表示该指令代码字长为 x 个字，当满足条件时执行该指令需 y 个机器周期，当不满足条件时只需 z 个机器周期，若采用延时方式则需 d 个机器周期。

1．算术指令

助记符指令	代 数 指 令	说　　明	字/周期
ABDST Xmem, Ymem	abdst（Xmem, Ymem）	计算两向量之差的绝对值	1/1
ABS src [, dst]	dst=\| src \|	累加器取绝对值	1/1
ADD Smem, src	src=src+Smem src+=Smem	操作数加到累加器	1/1
ADD Smem, TS, src	src=src+Smem<<TS src+=Smem<<TS	操作数移位后加到累加器	1/1
ADD Smem, 16, src [,dst]	dst=src+Smem<<16 dst+=Smem<<16	操作数左移 16 位加到累加器	1/1
ADD Smem [, SHIFT], src [, dst]	dst=src+Smem [<<SHIFT] dst+=Smem [<<SHIFT]	操作数移位后加到累加器	2/2
ADD Xmem, SHFT, src	src=src+Xmem<<SHFT src+=Xmem<<SHFT	操作数移位后加到累加器	1/1
ADD Xmem, Ymem, dst	dst=Xmem<<16+Ymem<<16	两操作数均左移 16 后加到累加器	1/1
ADD #lk [, SHFT], src [,dst]	dst=src+#lk[<<SHFT] dst+=#lk[<<SHFT]	16 位立即数移位后加到累加器	2/2
ADD #lk, 16, src [, dst]	dst=src+#lk<<16 dst+=#lk<<16	16 位立即数左移 16 位后加到累加器	2/2
ADD src [, SHIFT] [, dst]	dst=dst+src[<<SHIFT] dst+=src[<<SHIFT]	累加器移位后相加	1/1
ADD src, ASM [, dst]	dst=dst+src<<ASM dst+=src<<ASM	累加器按 ASM 移位后相加	1/1
ADDC Smem, src	src=src+Smem+CARRY src+=Smem+CARRY	操作数带进位加到累加器	1/1
ADDM #lk, Smem	Smem=Smem+#lk Smem+=#lk	16 位立即数加到存储器	2/2
ADDS Smem, src	src=src+uns（Smem） src+=uns（Smem）	无符号位扩展的加法	1/1
DADD Lmem, src [, dst]	dst=src+dbl（Lmem） dst+=dbl（Lmem）	32 位长数据存储器操作数加到源累加器中	1/1
DADST Lmem，dst	Dst=dadst（Lmem，T）	将 T 寄存器的值与 32 位长数据操作数相加，结果放入累器	1/1
DELAY Smem	delay（Smem）	存储器延时	1/1
DRSUB Lmem, src	src=dbl（Lmem）-src src=dual（Lmem）-src	从 32 位长数据存储器操作数中减去累加器内容，结果存放在累加器中	1/1
DSADT Lmem, dst	dst=dsadt（Lmem, T）	从 32 位长数据存储器操作数中减/加 T 寄存器的值，结果存放在累加器中	1/1

（续表）

助记符指令	代 数 指 令	说　　明	字/周期
DSUB Lmem, src	src=src-dbl（Lmem） src-=dbl（Lmem） src=src-dual（Lmem） src-=dual（Lmem）	从累加器中减去 32 位长数据存储器操作数的值	1/1
DSUBT Lmem, dst	dst=dbl（Lmem）-T dst=dual（Lmem）-T	从 32 位长数据存储器操作数中减去 T 寄存器的值	1/1
EXP src	T=exp（src）	求累加器值的指数	1/1
FIRS Xmem, Ymem, pmad	firs（Xmem, Ymem, pmad）	实现一个对称有限冲激响应滤波器	2/3
LMS Xmem, Ymem	lms（Xmem, Ymem）	求最小均方值	1/1
MAC[R] Smem, src	src=rnd（src+T*Smem）	T 寄存器值与操作数相乘后加到累加器，有后缀 R 时结果凑整	1/1
MAC[R] Xmem, Ymem, src [, dst]	dst=rnd（src+Xmem*Ymem）[,T=Xmem]	双操作数相乘后加到累加器，有后缀 R 时结果凑整	1/1
MAC #lk, src [, dst]	dst=src+T*#lk dst+=T*#lk	T 寄存器值乘以 16 位立即数后加到累加器	2/2
MAC Smem, #lk, src [, dst]	dst=src+Smem*#lk [, T=Smem] dst+=Smem*#lk [, T=Smem]	操作数乘以 16 位立即数后加到累加器	2/2
MACA[R] Smem [, B]	B=rnd（B+Smem*hi（A））[, T=Smem]	操作数乘以累加器 A 的高端（32~16 位）后加到累加器 B 中，有后缀 R 时结果凑整	1/1
MACA[R] T, src [, dst]	dst=rnd（src+T*hi（A））	T 寄存器值与累加器 A 高端（32~16 位）相乘后与源累加器相加	1/1
MACD Smem, pmad, src	macd（Smem, pmad, src）	操作数与程序存储器的值相乘后累加并延迟	2/3
MACP Smem, pmad, src	macp（Smem, pmad, src）	操作数乘以程序存储器的值后加到源累加器	2/3
MACSU Xmem, Ymem, src	src=src+uns（Xmem）*Ymem [, T=Xmem] src+=uns（Xmem）*Ymem [, T=Xmem]	无符号数与有符号数相乘后加到累加器	1/1
MAS[R] Smem, src	src=rnd（src-T*Smem）	源累加器减去 T 寄存器与操作数的乘积，有后缀 R 时结果凑整	1/1
MAS[R] Xmem, Ymem, src [, dst]	dst=rnd（src-Xmem*Ymem）[, T=Xmem]	源累加器减去双操作数的乘积，有后缀 R 时结果凑整	1/1
MASA Smem [,B]	B=B-Smem*hi（A）[,T=Xmem]	累加器 B 减去操作数与累加器 A 高端（32~16 位）的乘积	1/1
MASA[R] T, src [, dst]	dst=rnd（src-T*hi（A））	源累加器减去 T 寄存器与累加器 A 高端（32~16 位）的乘积，有后缀 R 时结果凑整	1/1
MAX dst	dst=max（A, B）	求累加器的最大值	1/1
MIN dst	dst=min（A, B）	求累加器的最小值	1/1
MPY[R] Smem, dst	dst=rnd（T*Smem）	T 寄存器与操作数相乘，有后缀 R 时结果凑整	1/1
MPY Xmem, Ymem, dst	dst=Xmem*Ymem [, T=Xmem]	两数据存储器操作数相乘	1/1
MPY Smem, #lk, dst	dst=Smem*#lk [, T=Smem]	操作数与 16 位立即数相乘	2/2
MPY #lk, dst	dst=T*#lk	T 寄存器与 16 位立即数相乘	2/2
MPYA dst	dst=T*hi（A）	T 寄存器与累加器 A 高端（32~16 位）相乘	1/1
MPYA Smem	B=Smem*hi（A）[, T=Smem]	单数据存储器操作数与累加器 A 高端（32~16 位）相乘	1/1
MPYU Smem, dst	dst=T*uns（Smem）	无符号数与 T 寄存器相乘	1/1
NEG src [, dst]	dst= -src	计算源累加器的二进制补码	1/1
NORM src [, dst]	dst=src<<TS dst=norm（src,TS）	对源累加器中带符号数归一化	1/1
POLY Smem	poly（Smem）	求多项式的值	1/1
RND src [, dst]	dst=rnd（src）	将 2^{15} 加到源累加器中进行凑整	1/1
SAT src	saturate（src）	将源累加器的值饱和为 32 位	1/1

（续表）

助记符指令	代 数 指 令	说 明	字/周期
SQDST Xmem, Ymem	sqdst（Xmem, Ymem）	计算两个向量距离的平方	1/1
SQUR Smem, dst	dst=Smem*Smem [, T=Smem] dst=square（Smem） [, T=Smem]	单数据存储器操作数的平方	1/1
SQUR A, dst	dst=hi（A）*hi（A） dst=square（hi（A））	计算累加器 A 高端（32~16 位）的平方	1/1
SQURA Smem, src	src=src+square（Smem） [, T=Smem] src+=square（Smem） [, T=Smem] src=src+ Smem*Smem [, T=Smem] src+ =Smem*Smem [, T=Smem]	操作数平方后加到源累加器	1/1
SQURS Smem, src	src=src-square（Smem） [, T=Smem] src-=square（Smem） [, T=Smem] src=src- Smem*Smem [, T=Smem] src- =Smem*Smem [, T=Smem]	源累加器减去操作数的平方	1/1
SUB Smem, src	src=src-Smem src- =Smem	从累加器中减去操作数	1/1
SUB Smem, TS, src	src=src-Smem<<TS src- =Smem<<TS	从累加器中减去移位后的操作数	1/1
SUB Smem, 16, src [, dst]	dst= src-Smem<<16 dst- =Smem<<16	从累加器中减去左移 16 位后的操作数	1/1
SUB Smem [, SHIFT], src [, dst]	dst=src-Smem[<<SHIFT] dst- =Smem[<<Smem]	从累加器中减去移位后的操作数	2/2
SUB Xmem, SHFT, src	src=src-Xmem<<SHFT src- =Xmem<<SHFT	从累加器中减去移位后的操作数	1/1
SUB Xmem, Ymem, dst	dst=Xmem<<16-Ymem<<16	两操作数均左移 16 位后相减	1/1
SUB #lk, 16, src [, dst]	dst=src-#lk[<<16] dst- =#lk[<<16]	累加器减去左移 16 位后的立即数	2/2
SUB #lk [, SHFT], src [, dst]	dst=src-#lk[<<SHFT] dst- =#lk[<<SHFT]	累加器减去移位后的 16 位立即数	2/2
SUB src , ASM [, dst]	dst=dst-src<<ASM dst- = src<<ASM	目的累加器减去按 ASM 移位后的源累加器	1/1
SUB src [, SHIFT], [, dst]	dst=dst-src<<SHIFT dst- = src<<SHIFT	目的累加器减去移位后的源累加器	1/1
SUBB Smem, src	src=src-Smem-BORROW src- =Smem-BORROW	从累加器中减去带借位的操作数	1/1
SUBC Smem, src	subc（Smem, src）	条件减法	1/1
SUBS Smem, src	src=src-uns（Smem） src- =uns（Smem）	无符号位扩展的减法	1/1

2．逻辑指令

助记符指令	代 数 指 令	说 明	字/周期
AND Smem, src	src=src & Smem src&=Smem	操作数与源累加器相与	1/1
AND #lk [, SHFT], src [, dst]	dst=src & #lk[<<SHFT] dst&=#lk[<<SHFT]	16 位立即数移位后和累加器相与	2/2
AND #lk 16, src [, dst]	dst=src & #lk<<16 dst&=#lk<<16	16 位立即数左移 16 位后和累加器相与	2/2
AND src [, SHIFT] [, dst]	dst=dst & src[<<SHIFT] dst&=src[<<SHIFT]	源累加器移位后和目的累加器相与	1/1
ANDM #lk, Smem	Smem=Smem & #lk Smem&=#lk	操作数与 16 位立即数相与	2/2

（续表）

助记符指令	代 数 指 令	说　　明	字/周期
BIT Xmem, BITC	TC=bit（Xmem, bit-code）	测试指定位	1/1
BITF Smem, #lk	TC=bitf（Smem, #lk）	测试单数据存储器 Smem 中指定的某些位	2/2
BITT Smem	TC=bitt（Smem）	测试由 T 寄存器指定的位	1/1
CMPL src [, dst]	dst= ~src	求累加器的反码	1/1
CMPM Smem, #lk	TC=（Smem= = #lk）	比较 16 位单数据存储器操作数 Smem 与 16 位立即数 lk 是否相等	2/2
CMPR CC, ARx	TC=（AR0= = ARx） TC=（AR0 > ARx） TC=（AR0 < ARx） TC=（AR0 != ARx）	辅助寄存器 ARx 与 AR0 比较	1/1
OR Smem, src	src=src \| Smem	操作数与源累加器相或	1/1
OR #lk [, SHFT], src [, dst]	dst=src \| #lk[<<SHFT] dst \| = #lk[<<SHFT]	16 位立即数移位后和累加器相或	2/2
OR #lk 16, src [, dst]	dst=src \| #lk<<16	16 位立即数左移 16 位后和累加器相或	2/2
OR src [, SHIFT] [, dst]	dst=dst \| src[<<SHIFT]	源累加器移位后和目的累加器相或	1/1
ORM #lk, Smem	Smem=Smem \| #lk	操作数和 16 位立即数相或	2/2
ROL src	src=src \\ CARRY	累加器带进位位循环左移	1/1
ROLTC src	roltc（src）	累加器带 TC 位循环左移	1/1
ROR src	src=src // CARRY	累加器带进位位循环右移	1/1
SFTA src, SHIFT [, dst]	dst=src<<C SHIFT	累加器算术移位	1/1
SFTC src	shiftc（src）	累加器条件移位	1/1
SFTL src, SHIFT [, dst]	dst=src<<< SHIFT	累加器逻辑移位	1/1
XOR Smem, src	src=src ^ Smem	操作数与源累加器相异或	1/1
XOR #lk [, SHFT], src[, dst]	dst=src ^ #lk[, SHFT]	16 位立即数移位后与累加器相异或	2/2
XOR #lk, 16, src [, dst]	dst= src ^ #lk<<16	16 位立即数左移 16 位后与累加器相异或	2/2
XOR src [, SHIFT] [, dst]	dst=dst ^ src[<<SHIFT]	源累加器移位后与目的累加器相异或	1/1
XORM #lk, Smem	Smem=Smem ^ #lk	操作数与 16 位立即数相异或	2/2

3．程序控制指令

助记符指令	代 数 指 令	说　　明	字/周期
B[D] pmad	goto pmad dgoto pmad	无条件转移，有后缀 D 时带延时	2/4[2]
BACC[D] src	goto src dgoto src	按累加器指定的地址转移，有后缀 D 时带延时	1/6[4]
BANZ[D]c pmad, Sind	if（Sind!=0） goto pmad if（Sind!=0） dgoto pmad	当辅助寄存器（ARP）不为 0 就转移，有后缀 D 时带延时	2/4,2[2]
BC[D] pmad, cond[, cond[, cond]]	if（cond[, cond[, cond]]） goto if（cond[, cond[, cond]]） dgoto	条件转移，有后缀 D 时带延时	2/5,3[3]
CALA[D] src	call src dcall src	按累加器规定的地址调用子程序，有后缀 D 时带延时	1/6[4]
CALL[D] pmad	call pmad dcall pmad	无条件调用子程序，有后缀 D 时带延时	2/4[2]
CC[D] pmad, cond[, cond[, cond]]	if（cond[, cond[, cond]]） call pmad if（cond[, cond[, cond]]） dcall pmad	有条件调用子程序，有后缀 D 时带延时	2/5,3[3]
FB[D] extpmad	far goto extpmad far dgoto extpmad	无条件长转移，有后缀 D 时带延时	2/4[2]

（续表）

助记符指令	代 数 指 令	说 明	字/周期
FBACC[D] src	far goto src far dgoto src	按累加器指定的地址长转移，有后缀 D 时带延时	1/6[4]
FCALA[D] src	far call src far dcall src	按累加器规定的地址长调用子程序，有后缀 D 时带延时	1/6[4]
FCALL[D] extpmad	far call extpmad far dcall extpmad	无条件长调用子程序，有后缀 D 时带延时	2/4[2]
FRAME K	SP=SP+K SP+=K	将一短立即数偏移 K 加到堆栈指针 SP 中	1/1
FRET[D]	far return far dreturn	长返回，有后缀 D 时带延时	1/6[4]
FRETE[D]	far return_enable far dreturn_enable	开中断，长返回，有后缀 D 时带延时	1/6[4]
IDLE K	idle（K）	强迫程序执行等待操作直至产生非屏蔽中断或复位操作	1/4
INTR K	int（K）	非屏蔽的软件中断，关闭其他可屏蔽中断	1/3
MAR Smem	mar（Smem）	修改辅助寄存器	1/1
NOP	nop	空操作	1/1
POPD Smem	Smem=pop（ ）	把数据从栈顶弹出到数据存储器	1/1
POPM MMR	MMR=pop（ ）	把数据从栈顶弹出到存储器映射寄存器	1/1
PSHD Smem	push（Smem）	把数据压入堆栈	1/1
PSHM Smem	push（MMR）	把数据存储器映射寄存器值压入堆栈	1/1
RC[D] cond[, cond[, cond]]	if（cond[, cond[, cond]]） return if（cond[, cond[, cond]]） dreturn	条件返回，有后缀 D 时带延时	1/5,3[3]
RESET	reset	软件复位	1/3
RET[D]	return dreturn	返回，有后缀 D 时带延时	1/5[3]
RETE[D]	return_enable dreturn_enable	打开中断并返回，有后缀 D 时带延时	1/5[3]
RETF[D]	return_fast dreturn_fast	打开中断并快速返回，有后缀 D 时带延时	1/5[3]
RPT Smem	repeat（Smem）	重复下条指令（Smem）+1 次	1/1
RPT #K	repeat（#K）	重复下条指令 K+1 次	1/1
RPT #lk	repeat #lk	重复下条指令 lk+1 次	1/1
RPTB[D] pmad	blockrepeat（pmad） dblockrepeat（pmad）	块重复指令，有后缀 D 时带延时	2/4[2]
RPTZ dst, #lk	repeat（#lk）, dst=0	累加器清零并重复下条指令 lk+1 次	2/2
RSBX N, SBIT	SBIT=0 ST（N, SBIT）=0	将状态寄存器 ST0 和 ST1 的特定位清零	1/1
SSBX N, SBIT	SBIT=1 ST（N, SBIT）=1	将状态寄存器 ST0 和 ST1 的特定位置 1	1/1
TRAP K	trap（K）	软件中断	1/3
XC n, cond[, cond[, cond]]	if（cond[, cond[, cond]]） execute（n）	条件执行	1/1

4．装载和存储指令

助记符指令	代 数 指 令	说 明	字/周期
CMPS src, Smem	cmps（src, Smem）	比较源累加器高端（31~16 位）和低端（15~0）两个 16 位二进制补码值的大小，并存储最大值	1/1

（续表）

助记符指令	代 数 指 令	说　　明	字/周期
DLD, Lmem, dst	dst=dbl（Lmem） dst=dual（Lmem）	将 32 位的长操作数装入目的累加器	1/1
DST src, Lmem	dbl（Lmem）=src dual（Lmem）=src	将源累加器值存入 32 位长数据存储器单元	1/2
LD Smem, dst	dst=dbl（Lmem） dst=dual（Lmem）	将操作数装入累加器	1/1
LD Smem, TS, dst	dst=Smem<<TS	操作数移位后装入累加器	1/1
LD Smem, 16, dst	dst=Smem<<16	操作数左移 16 位后装入累加器	1/1
LD Smem [, SHIFT], dst	dst=Smem[<<SHIFT]	操作数移位后装入累加器	2/2
LD Xmem, SHFT, dst	dst=Xmem<<SHFT	操作数移位后装入累加器	1/1
LD #K, dst	dst= #K	短立即数装入累加器	1/1
LD #lk [, SHFT], dst	dst= #lk[<<SHFT]	长立即数移位后装入累加器	2/2
LD #lk, 16, dst	dst= #lk<<16	长立即数左移 16 位后装入累加器	2/2
LD src, ASM [, dst]	dst= src<<ASM	源累加器按 ASM 移位后装入目的寄存器	1/1
LD src [, SHIFT] [, dst]	dst=src[<<SHIFT]	源累加器移位后装入目的累加器	1/1
LD Smem, T	T=Smem	操作数装入 T 寄存器	1/1
LD Smem, DP	DP=Smem	9 位操作数装入 DP	1/3
LD #k9, DP	DP= #k9	9 位立即数装入 DP	1/1
LD #k5, ASM	ASM= #k5	5 位立即数装入 ASM	1/1
LD #k3, ARP	ARP= #k3	3 位立即数装入 ARP	1/1
LD Smem, ASM	ASM=Smem	5 位操作数装入 ASM	1/1
LD Xmem, dst ‖ MAC[R] Ymem [, dst]	dst=Xmem[<<16] ‖ dst_=rnd（dst_+T*Ymem）	装载累加器，并行执行乘累加运算，有后缀 R 时结果凑整	1/1
LD Xmem, dst ‖ MAS[R] Ymem [, dst]	dst=Xmem[<<16] ‖ dst_=rnd（dst_-T*Ymem）	装载累加器，并行执行乘累减运算，有后缀 R 时结果凑整	1/1
LDM MMR, dst	dst=MMR	将存储器映射寄存器值装入累加器	1/1
LDR Smem, dst	dst=rnd（Smem）	操作数凑整后装入累加器	1/1
LDU Smem, dst	dst=uns（Smem）	无符号操作数装入累加器	1/1
LTD Smem	ltd（Smem）	操作数装入 T 寄存器并延迟	1/1
SACCD src, Xmem, cond	if（cond） Xmem=hi（src）<<ASM	条件存储累加器的值	1/1
SRCCD Xmem, cond	if（cond） Xmem=BRC	条件存储块重复计数器的值	1/1
ST T, Smem	Smem=T	存储 T 寄存器的值	1/1
ST TRN, Smem	Smem=TRN	存储 TRN 寄存器的值	1/1
ST #lk, Smem	Smem=#lk	存储长立即数	2/2
STH src, Smem	Smem=hi（src）	源累加器高端（31~16 位）存放到数据存储器	1/1
STH src, ASM, Smem	Smem=hi（src）<<ASM	源累加器高端（31~16 位）按 ASM 移位后存储	1/1
STH src, SHFT, Xmem	Xmem=hi（src）<<SHFT	源累加器高端（31~16 位）移位后存储	1/1
STH src [, SHIFT], Smem	Smem=hi（src）<<SHIFT	源累加器高端（31~16 位）移位后存储	2/2
ST src, Ymem ‖ ADD Xmem, dst	Ymem=hi（src）[<<ASM] ‖ dst=dst_+Xmem<<16	存储累加器，并行执行加法运算	1/1
ST src, Ymem ‖ LD Xmem, dst	Ymem=hi（src）[<<ASM] ‖ dst=Xmem<<16	存储累加器，并行执行装载累加器	1/1
ST src, Ymem ‖ LD Xmem, T	Ymem=hi（src）[<<ASM] ‖ T=Xmem	存储累加器，并行执行装载 T 寄存器	1/1
ST src, Ymem ‖ MAC[R] Xmem, dst	Ymem=hi（src）[<<ASM] ‖ dst=rnd（dst+T*Xmem）	存储累加器，并行执行乘累加运算，有后缀 R 时结果凑整	1/1
ST src, Ymem ‖ MAS[R] Xmem, dst	Ymem=hi（src）[<<ASM] ‖ dst=rnd（dst-T*Xmem）	存储累加器，并行执行乘累减运算，有后缀 R 时结果凑整	1/1

（续表）

助记符指令	代 数 指 令	说　　明	字/周期
ST src, Ymem \|\| MPY Xmem, dst	Ymem=hi（src）[<<ASM] \|\| dst=T*Xmem	存储累加器，并行执行乘法运算	1/1
ST src, Ymem \|\| SUB Xmem, dst	Ymem=hi（src）[<<ASM] \|\| dst=Xmem<<16 - dst	存储累加器，并执行减法运算	1/1
STL src, Smem	Smem=src	将累加器低端（15~0 位）存放到数据存储器中	1/1
STL src, ASM, Smem	Smem=src<<ASM	源累加器按 ASM 移位后低端（15~0 位）存放到数据存储器	1/1
STL src, SHFT, Xmem	Xmem=src<<SHFT	源累加器按 SHFT 移位后低端（15~0 位）存放到数据存储器	1/1
STL src [, SHIFT], Smem	Smem=src<<SHIFT	源累加器按 SHIFT 移位后低端（15~0 位）存放到数据存储器	2/2
STLM src, MMR	MMR=src mmr（MMR）=src	源累加器存入存储器映射寄存器	1/1
STM #lk, MMR	MMR= #lk mmr（MMR）= #lk	长立即数存入存储器映射寄存器	2/2
STRCD Xmem, cond	if（cond） Xmem=T	条件存储 T 寄存器的值	1/1
MVDD Xmem, Ymem	Ymem=Xmem	在数据存储器内部传送数据	1/1
MVDK Smem, dmad	data（dmad）=Smem	数据存储器目的地址寻址的数据传送	2/2
MVDM dmad, MMR	MMR=data（dmad） mmr（MMR）=data（dmad）	从数据存储器向映射寄存器传送数据	2/2
MVDP Smem, pmad	prog（pmad）=Smem	从数据存储器向程序存储器传送数据	2/2
MVKD dmad, Smem	Smem=data（dmad）	数据存储器源地址寻址的数据传送	2/4
MVMD MMR, dmad	data（dmad）=MMR data（dmad）=mmr（MMR）	从存储器映射寄存器向数据存储器传送数据	2/2
MVMM MMRx, MMRy	MMRy=MMRx mmr（MMRy）=mmr（MMRx）	从存储器映射寄存器内部传送数据	1/1
MVPD pmad, Smem	Smem=prog（pmad）	从程序存储器向数据存储器传送数据	2/3
PORTR PA, Smem	Smem=port（PA）	从 PA 口读入数据	2/2
PORTW Smem, PA	port（PA）=Smem	向 PA 口输出数据	2/2
READA Smem	Smem=prog（A）	按累加器 A 内容寻址，读程序存储器值并存入数据存储器	1/5
WRITA Smem	prog（A）=Smem	将数据写入按累加器 A 寻址的程序存储器单元	1/5

附录 B　TI 格式文件转化为二进制文件

```
/* 文件名 T2B54.C
（1）TI 格式文件转化为二进制文件
（2）16 位 BOOT 方式
（3）本程序适用于程序和初始化数据不超过 64K 字节
（4）使用方法:  T2B54 tiname
    输入文件为TI文件格式，不用输入扩展名；
    输出文件为具有BOOT功能的二进制文件，文件名为 tiname.bin */
    #include <stdio.h>
    #include <string.h>
    #include <stdlib.h>
    #include <math.h>

    //global function define
    int value（char ch）;
    void printHelpMessage（）;

    void main（int argc,char *argv[]）
    {
        long int i,j;
        char   fname[65],gname[65],inpstr[256];
        unsigned char   progid[10];
        long int cur_addr,first_addr,last_addr;
        short temp,n;
        FILE *fpin,*fpout;

        switch （argc）
        {
        case 1:
            printHelpMessage（）;
            exit（1）;
        case 2:
            //copyright statement
            printf（"TI-Tagged format file to BINary format file converter Ver 1.01\n"）;
            printf（"                    For   TMS320C54x\n"）;
            printf（"Writen by Zhang Xiongwei,PLA UST,Nanjing,China,Aug.,2007 \n"）;
            printf（"All Rights Reserved \n"）;
            strcpy（fname,argv[1]）;
            strcpy（gname,fname）;
            strcat（fname,".tag"）;
            strcat（gname,".bin"）;
            printf（"%s    \n%s \n",fname,gname）;
            break;
```

```
default:
        printHelpMessage（）；
        exit（1）；
}

if（（fpin=fopen（fname,"rt"））==NULL）
{
        printf（"Could not open input file\n"）；
        exit（0）；
}

if（（fpout=fopen（gname,"wb"））==NULL）
{
        printf（"Could not open output file\n"）；
        exit（0）；
}
fseek（fpout,16,SEEK_SET）；

first_addr=32767L;
last_addr=0;
for （;;）
{
        if（fgets（inpstr,128,fpin）==NULL）
                break;
        if （inpstr[0]==':'）
                break;

        i=0;
        n = strlen（inpstr）；
        while （i<n）
        {
                if （inpstr[i]=='K'）
                {
                        i=i+5;
                        for （j=i;j<i+8;j++）
                                progid[j-i]=inpstr[j];
                        progid[8]=0;
                        i=i+8;
                        printf（"Program Name is %s \n",progid）；
                }else
                        if （inpstr[i]=='9'）
                        {
                                cur_addr=（long）value（inpstr[i+1]）*4096+（long）value（inpstr[i+2]）*256
                                        +（long）value（inpstr[i+3]）*16+（long）value（inpstr[i+4]）；

                                printf（"The current address is %lx H \n",cur_addr）；
                                if （first_addr>cur_addr）      first_addr=cur_addr;
                                if （last_addr<cur_addr）       last_addr=cur_addr;
```

```
                    i=i+5;
                }else
                    if （inpstr[i]=='B'）
                    {
                        temp = （（value（inpstr[i+1]）*16+value（inpstr[i+2]））<<8）

                            +value（inpstr[i+3]）*16+value（inpstr[i+4]）;
                        fwrite（&temp,sizeof（short）,1,fpout）;
                        cur_addr++;
                        i=i+5;
                        if （last_addr<cur_addr）
                            last_addr=cur_addr;
                    }else
                        if （inpstr[i]=='7'）
                            break;
                        else
                            if （inpstr[i]=='F'）
                                break;
                            else
                                break;
        }
}

/*c54 nop = 0f495h*/
temp = （short）0xf495;
for（i=cur_addr-first_addr+8;i<32767;i++）
{
    fwrite（&temp,sizeof（short）,1,fpout）;
}
/*address 0xffff contains source address=8000h*/
if（i==32767）
{
    temp = （short）0x8000;
    fwrite（&temp,sizeof（short）,1,fpout）;
}
printf（"The   first address is %lx H \n",first_addr）;
printf（"The   last address is %x H \n",last_addr）;

/* write the loader header */
fseek（fpout,0,SEEK_SET）;
temp = （short）0x10aa;          /*10AAh:识别字: 16 位 boot 方式 */
fwrite（&temp,sizeof（short）,1,fpout）;
temp = （short）0x7fff;          /*SWWSR=07fffh: 插入 7 个等待周期*/
fwrite（&temp,sizeof（short）,1,fpout）;
temp = （short）0xf800;          /*BSCR=0f800h*/
fwrite（&temp,sizeof（short）,1,fpout）;
temp = （short）0x0000;          /*entry XPC=0000h*/
fwrite（&temp,sizeof（short）,1,fpout）;
temp = （short）0x7c00;          /*entry address=7c00h user boot program entry*/
```

```
        fwrite（&temp,sizeof（short）,1,fpout）;
        temp = （short）0x0100;          /*N=100h*/
        fwrite（&temp,sizeof（short）,1,fpout）;
        temp = （short）0x0000;          /*code dest XPC=0000h*/
        fwrite（&temp,sizeof（short）,1,fpout）;
        temp = （short）0x7c00;          /*code dest address=7c00h*/
        fwrite（&temp,sizeof（short）,1,fpout）;

        fclose（fpin）;
        fclose（fpout）;
        printf（"TMS320C54x-TAG to BIN convertion completed!\n "）;
        return;
}

int value（char ch）
{
        switch（ ch）
        {
        case '0':return（0）;
        case '1':return（1）;
        case '2':return（2）;
        case '3':return（3）;
        case '4':return（4）;
        case '5':return（5）;
        case '6':return（6）;
        case '7':return（7）;
        case '8':return（8）;
        case '9':return（9）;
        case 'A':return（10）;
        case 'B':return（11）;
        case 'C':return（12）;
        case 'D':return（13）;
        case 'E':return（14）;
        case 'F':return（15）;
        }
        return 0;
}

void printHelpMessage（）
{
        printf（"\n"）;
        printf（"Usage : T2B54 TI-Tagged_file BIN_file\n"）;
       printf（"\n"）;
        printf（"TI-Tagged format file to BINary format file converter Ver 1.01\n"）;
        printf（"                    For    TMS320C54x\n"）;
        printf（"Writen by Zhang Xiongwei,PLA UST,Nanjing,China,Aug.,2007 \n"）;
        printf（"All Rights Reserved \n"）;
        exit（1）;
}
```

附录 C　8 位 μ 律/16 位线性互换的 C 语言程序

```
/* This routine converts from linear to ulaw
      线性到 u 律转换子程序
      输入：16 位线性样值
      输出：8 位 u 律样值　*/
#define ZEROTRAP      /* turn on the trap as per the MIL-STD */
#define BIAS 0x84     /* define the add-in bias for 16 bit samples */
#define CLIP 32635
unsigned char     linear2ulaw（int sample）
{
     static short exp_lut[256] = {0,0,1,1,2,2,2,2,3,3,3,3,3,3,3,3,
                                  4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,
                                  5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
                                  5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
                                  6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,
                                  6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,
                                  6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,
                                  6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,
                                  7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
                                  7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
                                  7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
                                  7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
                                  7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
                                  7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
                                  7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
                                  7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7};
     int sign, exponent, mantissa;
     unsigned char ulawbyte;
      /** get the sample into sign-magnitude ** /
     sign =  （sample >> 8）   & 0x80;        /* set aside the sign */
     if  （sign != 0）  sample = -sample;      /* get magnitude */
     if  （sample > CLIP）  sample = CLIP;   /* clip the magnitude */
     /** convert from 16 bit linear to ulaw **/
     sample = sample + BIAS;
     exponent = exp_lut[（sample>>7）  & 0xFF];
     mantissa =  （sample >>  （exponent+3））  & 0x0F;
     ulawbyte = ~（sign |  （exponent << 4）  | mantissa）;
#ifdef ZEROTRAP
     if  （ulawbyte == 0  ）  ulawbyte = 0x02;   /* optional CCITT trap */
```

```
#endif
    /** return the result **/
    return（ulawbyte）;
}

/* This routine converts from ulaw to 16 bit linear
     u 律到线性转换子程序
     * 输入: 8 位 u 律样值
     * 输出: 16 位线性样值 */
int ulaw2linear（ulawbyte）
unsigned char ulawbyte;
{
    static int exp_lut[8]={0,132,396,924,1980,4092,8316,16764};
    int sign, exponent, mantissa, sample;
    ulawbyte = ~ulawbyte;
    sign = （ulawbyte & 0x80）;
    exponent = （ulawbyte >> 4） & 0x07;
    mantissa = ulawbyte & 0x0F;
    sample = exp_lut[exponent] + （mantissa << （exponent+3））;
    if （sign != 0） sample = -sample;
    return（sample）;
}
```

附录 D　μ 律到线性变换表

```
float   ulaw_tab[256]=
    {-7774.0,-7518.0,-7262.0,-7006.0,-6750.0,-6494.0,-6238.0,-5982.0,
     -5726.0,-5470.0,-5214.0,-4958.0,-4702.0,-4446.0,-4190.0,-3998.0,
     -3870.0,-3742.0,-3614.0,-3486.0,-3358.0,-3230.0,-3102.0,-2974.0,
     -2846.0,-2718.0,-2590.0,-2462.0,-2334.0,-2206.0,-2078.0,-1982.0,
     -1918.0,-1854.0,-1790.0,-1726.0,-1662.0,-1598.0,-1534.0,-1470.0,
     -1406.0,-1342.0,-1278.0,-1214.0,-1150.0,-1086.0,-1022.0, -974.0,
      -942.0, -910.0,   -878.0, -846.0, -814.0,   -782.0, -750.0, -718.0,
      -686.0, -654.0,   -622.0, -590.0, -558.0,   -526.0, -494.0, -470.0,
      -454.0, -438.0,   -422.0, -406.0, -390.0,   -374.0, -358.0, -342.0,
      -326.0, -310.0,   -294.0, -278.0, -262.0,   -246.0, -230.0, -218.0,
      -210.0, -202.0,   -194.0, -186.0, -178.0,   -170.0, -162.0, -154.0,
      -146.0, -138.0,   -130.0, -122.0, -114.0,   -106.0,   -98.0,   -92.0,
       -88.0,   -84.0,   -80.0,   -76.0,   -72.0,   -68.0,   -64.0,   -60.0,
       -56.0,   -52.0,   -48.0,   -44.0,   -40.0,   -36.0,   -32.0,   -31.0,
       -29.0,   -27.0,   -25.0,   -23.0,   -21.0,   -19.0,   -17.0,   -15.0,
       -13.0,   -11.0,    -9.0,    -7.0,    -5.0,    -3.0,    -1.0,     0.0,
      7776.0, 7520.0, 7264.0, 7008.0, 6752.0, 6496.0, 6240.0,   5984.0,
      5728.0, 5472.0, 5216.0, 4960.0, 4704.0, 4448.0, 4192.0,   4000.0,
      3872.0, 3744.0, 3616.0, 3488.0, 3360.0, 3232.0, 3104.0,   2976.0,
      2848.0, 2720.0, 2592.0, 2464.0, 2336.0, 2208.0, 2080.0,   1984.0,
      1920.0, 1856.0, 1792.0, 1728.0, 1664.0, 1600.0, 1536.0,   1472.0,
      1408.0, 1344.0, 1280.0, 1216.0, 1152.0, 1088.0, 1024.0,    976.0,
       944.0,   912.0,   880.0,   848.0,   816.0,   784.0,   752.0, 720.0,
       688.0,   656.0,   624.0,   592.0,   560.0,   528.0,   496.0, 472.0,
       456.0,   440.0,   424.0,   408.0,   392.0,   376.0,   360.0, 344.0,
       328.0,   312.0,   296.0,   280.0,   264.0,   248.0,   232.0, 220.0,
       212.0,   204.0,   196.0,   188.0,   180.0,   172.0,   164.0, 156.0,
       148.0,   140.0,   132.0,   124.0,   116.0,   108.0,   100.0,   94.0,
        90.0,    86.0,    82.0,    78.0,    74.0,    70.0,    66.0,   62.0,
        58.0,    54.0,    50.0,    46.0,    42.0,    38.0,    34.0,   33.0,
        31.0,    29.0,    27.0,    25.0,    23.0,    21.0,    19.0,   17.0,
        15.0,    13.0,    11.0,     9.0,     7.0,     5.0,     3.0,    0.0};
```

附录 E 缩写词的英文对照

ACC	ACCumulator
A/D	Analog to Digital Transform
ALU	Arithmetic and Logic Unit
API	Application Program Interface
ARAU	Auxiliary Register Arithmetic Unit
ARP	Auxiliary Register Pointer
ARB	Auxiliary Register pointer Buffer
ASIC	Application-Specific Integrated Circuit
AT&T	American Telephone and Telegraph company
BGA	Ball Grid Array
BIOS	Basic Input Output System
BOPS	Billion Operations Per Second
BSP	Buffered Serial Port
CALU	Central Arithmetic Logic Unit
CAN	Controller Area Network
CCS	Code Composer Studio
CD	Compact Disk
CMOS	Complementary Metal-Oxide Semiconductor
CPLD	Complex Programmable Logic Device
CQFP	Ceramic Quad Flat Package
CPU	Central Processing Unit
D/A	Digital to Analog Transform
DFT	Discrete Fourier Transform
DIP	Dual In-line Package
DARAM	Dual-Access RAM
DRAM	Dynamic RAM
DSP	Digital Signal Processing
DMA	Direct Memory Access
DVD	Digital Video Disk
EDA	Electronic Design Automation
EMC	ElectroMagnetic Compatibility
EPLD	Erasable Programmable Logic Device
EPROM	Erasable PROM
FFT	Fast Fourier Transform
FIFO	First In First Out
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
GEL	General Extension Language
HLL	High Level Programming Language

HPI	Host Port Interface
IC	Integrated Circuits
IIR	Infinite Impulse Response
IPTR	Interrupt Vector Pointer
ISR	Interrupt Service Routine
JTAG	Joint Test Action Group
LMS	Least Mean Square
LQFP	Low-Profile Quad Flat Pack
MAC	Multiplier and Accumulator
McBSP	Multi-channel Buffered Serial Port
MFLOPS	Million Floating-point Operations Per Second
MIPS	Million Instructions Per Second
MMR	Memory Mapped Register
MOPS	Million Operations Per Second
MCU	Micro-Computer Unit
MPSD	Modular Port Scan Device
MPU	Micro-Processor Unit
NMOS	N-channel Metal Oxide Semiconductor
OLE	Object Link Embedded
OMAP	Open Multimedia Application Platform
OSPA	Open-Signal Processing Architecture
PCB	Printed Circuit Board
PCM	Pulse Coded Modulation
PIE	Peripheral Interrupt Expansion
PGA	Pin Grid Array
PLD	Programmable Logic Device
PLL	Phase-Locked Loop
PLCC	Plastic Leaded Chip Carrier
PLU	Parallel Logic Unit
PMST	Processor Mode Status
PQFP	Plastic Quad Flat Pack
PROM	Programmable Read Only Memory
PWM	Pulse Width Modulation
QEP	Quadrature Encoder Pulse
QFP	Quad Flat Package
RISC	Reduced Instruction Set Computer
RAM	Random Access Memory
ROM	Read Only Memory
RTDX	Real Time Data Exchange
SARAM	Single-Access RAM
SCI	Serial Communications Interface
SOP	Small Outline Pack
SPI	Serial Peripheral Interface

SRAM	Static RAM
SWWSR	Software Wait-State Register
SWCR	Software Wait-State Control Register
TI	Texas Instruments
TQFP	Thin Quad Flat Pack
TTL	Transistor-Transistor Logic
UART	Universal Asynchronous Receiver and Transmitter
VCD	Video Compact Disk
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

参 考 文 献

［1］张雄伟．DSP芯片的原理与开发应用．北京：电子工业出版社，1997

［2］张雄伟，曹铁勇．DSP芯片的原理与开发应用（第2版）．北京：电子工业出版社，2000

［3］张雄伟，陈亮，徐光辉．DSP芯片的原理与开发应用（第3版）．北京：电子工业出版社，2003

［4］TMS320C54x DSP Reference Set, Volume 1: CPU and Peripherals, SPRU131G, Texas Instruments, 2001

［5］TMS320C54x DSP Reference Set, Volume 2: Mnemonic Instruction Set, SPRU172C, Texas Instruments, 2001

［6］TMS320C54x DSP Reference Set, Volume 3: Algebraic Instruction Set, SPRU179C, Texas Instruments, 2001

［7］TMS320C54x DSP Reference Set, Volume 4: Applications Guide, SPRU173, Texas Instruments, 2001

［8］TMS320C54x DSP Reference Set, Volume 5: Enhanced Peripherals, SPRU302, Texas Instruments, 1999

［9］TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User's Guide, SPRU018, Texas Instruments

［10］TMS320C2x/C2xx/C5x Optimizing C Compiler User's Guide, SPRU024, Texas Instruments

［11］TMS320C20x User's Guide, SPRU127, Texas Instruments

［12］TMS320F/C24x DSP Controllers Reference Guide:CPU and Instruction Set, SPRU160, Texas Instruments

［13］TMS320F/C240 DSP Controllers Peripheral Library, SPRU161, Texas Instruments

［14］TMS320F240 DSP Controllers Evaluation Module Tech Ref, SPRU248, Texas Instruments

［15］TMS320F243/F241/C242 DSP Controllers System and Peripherals, SPRU276, Texas Instruments

［16］TMS320F20x/F24x DSP Embedded Flash Memory Tech Ref, SPRU282, Texas Instruments

［17］TMS320LF/LC240xA DSP Controllers System and Peripherals, SPRU357, Texas Instruments

［18］TMS320LF2407, TMS320LF2406, TMS320LF2402 DSP Controllers, SPRS094, Texas Instruments

［19］TMS320C240, TMS320F240 DSP Controllers, SPRS042, Texas Instruments

［20］Code Composer Studio Getting Started Guide, SPRU509, Texas Instruments

［21］Code Composer Studio White Paper, SPRU520, Texas Instruments

［22］Code Composer Studio's Command Window, SPRU601, Texas Instruments

［23］TMS320C54CST Bootloader Technical Reference, SPRA853A, Texas Instruments

［24］Using the TMS320VC5501/C5502 Bootloader, SPRA911C, Texas Instruments

［25］TMS320C6000 Boot Mode and Emulation Reset , SPRA978, Texas Instruments

［26］Using the TMS320C672x Bootloader, SPRAA69C, Texas Instruments

［27］Using the TMS320C642x Bootloader, SPRAAK5, Texas Instruments

［28］TMS320F281x Boot ROM Serial Flash Programming, SPRAAQ2, Texas Instruments

［29］TMS320C3x User’s Guide, SPRU031F, Texas Instruments

［30］TMS320C3x/C4x Optimizing C Compiler User’s Guide, SPRU034H, Texas Instruments

［31］TMS320C3x/C4x Assembly Language Tools, SPRU035C, Texas Instruments

［32］TMS320x281x DSP Multichannel Buffered Serial Port （McBSP） Reference Guide, SPRU061C, Texas Instruments

［33］TMS320C4x User’s Guide, SPRU063C, Texas Instruments
［34］TMS320C54x C Source Debugger User’s Guide, SPRU099D, Texas Instruments
［35］TMS320C54x Assembly Language Tools User’s Guide, SPRU102F, Texas Instruments
［36］TMS320C54x Optimizing C/C++ Compiler User’s Guide, SPRU103G, Texas Instruments
［37］TMS320C6000 CPU and Instruction Set, SPRU189G, Texas Instruments
［38］TMS320C6000 DSP Peripherals Overview, SPRU190O, Texas Instruments
［39］TMS320C6000 Technical Briefm, SPRU197D, Texas Instruments
［40］TMS320C6000 Programmer’s Guide, SPRU198I, Texas Instruments
［41］TMS320C6000 DSP External Memory Interface （EMIF） Reference Guide, SPRU266E, Texas Instruments
［42］TMS320C55x Assembly Language Tools User’s Guide, SPRU280G, Texas Instruments
［43］TMS320C55x Optimizing C/C++ Compiler User’s Guide, SPRU281E, Texas Instruments
［44］TMS320C548/C549 Bootloader and ROM Code Contents Technical Reference, SPRU288A, Texas Instruments
［45］TMS320C54x DSP Functional Overview, SPRU307A, Texas Instruments
［46］TMS320C55x DSP Functional Overview, SPRU312, Texas Instruments
［47］TMS320C55x DSP Peripherals Overview Reference Guide, SPRU317F, Texas Instruments
［48］Code Composer Studio User’s Guide, SPRU328B, Texas Instruments
［49］TMS320C55x DSP CPU Reference Guide, SPRU371F, Texas Instruments
［50］TMS320C55x DSP Mnemonic Instruction Set Reference Guide, SPRU374G, Texas Instruments
［51］TMS320C55x DSP Algebraic Instruction Set Reference Guide, SPRU375G, Texas Instruments
［52］TMS320C55x DSP Programmer’s Guide, SPRU376A, Texas Instruments
［53］TMS320C55x Technical Overview, SPRU393, Texas Instruments
［54］TMS320C64x Technical Overview, SPRU395B, Texas Instruments
［55］TMS320C54x Chip Support Library API Reference Guide, SPRU420, Texas Instruments
［56］TMS320C55x DSP Library Programmer’s Reference, SPRU422A, Texas Instruments
［57］TMS320C28x DSP CPU and Instruction Set Reference Guide, SPRU430D, Texas Instruments
［58］TMS320C54x DSP Programmer’s Guide, SPRU538, Texas Instruments
［59］TMS320C620x/C670x DSP Program and Data Memory Controller/Direct Memory Access （DMA） Controller Reference Guide, SPRU577A, Texas Instruments
［60］TMS320C6000 DSP Host Port Interface （HPI） Reference Guide, SPRU578C, Texas Instruments
［61］TMS320C6000 DSP Multichannel Buffered Serial Port （McBSP） Reference Guide, SPRU580G, Texas Instruments
［62］TMS320VC5509/5510 DSP Direct Memory Access （DMA） Controller Reference Guide, SPRU587, Texas Instruments
［63］TMS320VC5510 DSP Host Port Interface （HPI） Reference Guide, SPRU588, Texas Instruments
［64］TMS320VC5510 DSP External Memory Interface （EMIF） Reference Guide, SPRU590, Texas Instruments
［65］TMS320VC5501/5502/5509/5510 DSP Multichannel Buffered Serial Port （McBSP） Reference Guide, SPRU592A, Texas Instruments
［66］TMS320VC5501/5502 DSP Direct Memory Access （DMA） Controller Reference Guide, SPRU613A,

Texas Instruments
［67］TMS320VC5509 DSP Host Port Interface （HPI） Reference Guide, SPRU619, Texas Instruments
［68］TMS320VC5501/5502 DSP Host Port Interface （HPI） Reference Guide, SPRU620A, Texas Instruments
［69］TMS320VC5501/5502 DSP External Memory Interface （EMIF） Reference Guide, SPRU621B, Texas Instruments
［70］TMS320C620x/C670x DSP Boot Modes and Configuration Reference Guide, SPRU642, Texas Instruments
［71］TMS320C55x DSP CPU Programmer’s Reference Supplement, SPRU652B, Texas Instruments
［72］TMS320VC5509 DSP External Memory Interface （EMIF） Reference Guide, SPRU670, Texas Instruments
［73］TMS320C672x DSP Peripherals Overview Reference Guide, SPRU723B, Texas Instruments
［74］TMS320C62x DSP CPU and Instruction Set Reference Guide, SPRU731, Texas Instruments
［75］TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide, SPRU732D, Texas Instruments
［76］TMS320C67x/C67x+ DSP CPU and Instruction Set Reference Guide, SPRU733A, Texas Instruments
［77］TMS320C642x DSP Host Port Interface （HPI） User's Guide, SPRUEM9, Texas Instruments
［78］TMS320C642x DSP Multichannel Buffered Serial Port （McBSP） Interface User's Guide, SPRUEN2B, Texas Instruments
［79］TMS320C28x Floating Point Unit and Instruction Set Reference Guide, SPRUEO2, Texas Instruments
［80］TMS320x2833x Direct Memory Access （DMA） Module Reference Guide, SPRUFB8, Texas Instruments
［81］何苏勤，王忠勇．TMS320C2000系列DSP原理及实用技术．北京：电子工业出版社，2003
［82］江思敏等．TMS320LF240x DSP硬件开发教程．北京：机械工业出版社，2003
［83］清源科技．TMS320LF240x DSP应用程序设计教程．北京：机械工业出版社，2003
［84］胡广书，崔子经．数字信号处理－一门正在蓬勃发展的新学科．电子科技导报，1995年第1期
［85］邹传云，敖发良，张得琨．DSP微处理器的基础特征．电子科技导报，1995年第11期
［86］王天祥．数字信号处理器（DSP）及其支持芯片的选择．DSP开发与应用，1996年第1期
［87］张雄伟．COFF－TMS320C31采用的目标文件格式．军事通信技术，1996年第3期
［88］张雄伟，陈亮，徐光辉．DSP 集成开发与应用实例．北京：电子工业出版社，2002
［89］〔美〕Texas Instruments Incorporated著，田黎育等编译．TMS320C6000系列DSP编程工具与指南．北京：清华大学出版社，2006
［90］〔美〕Texas Instruments Incorporated著，李海森等编译．TMS320C55x系列DSP指令系统、开发工具与编程指南．北京：清华大学出版社，2007
［91］〔英〕Andrew Bateman、iain paterson-Stephens著，陈健等译．DSP算法、应用与设计．北京：机械工业出版社，中信出版社，2004
［92］李广弟，朱月秀，冷祖祁．单片机基础（第 3 版）．北京航空航天大学出版社，2007
［93］任丽香，马淑芬，李方慧．TMS320C6000 系列 DSPs 的原理与应用．北京：电子工业出版社，2000